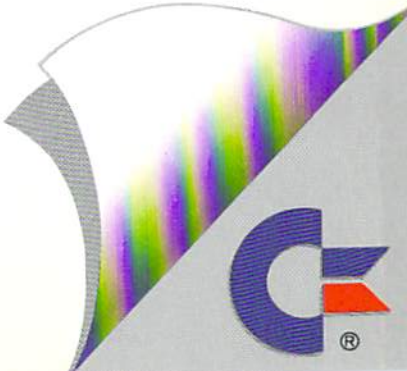
COMMODORE®

1581™

DISK DRIVE

user's guide



USER'S MANUAL STATEMENT

WARNING: This equipment has been certified to comply with the limits for a Class B computing device, pursuant to subpart J of Part 15 of the Federal Communications Commission's rules, which are designed to provide reasonable protection against radio and television interference in a residential installation. If not installed properly, in strict accordance with the manufacturer's instructions, it may cause such interference. If you suspect interference, you can test this equipment by turning it off and on. If this equipment does cause interference, correct it by doing any of the following:

- Reorient the receiving antenna or AC plug.
- Change the relative positions of the computer and the receiver.
- Plug the computer into a different outlet so the computer and receiver are on different circuits.

CAUTION: Only peripherals with shield-grounded cables (computer input-output devices, terminals, printers, etc.), certified to comply with Class B limits, can be attached to this computer. Operation with non-certified peripherals is likely to result in communications interference.

Your house AC wall receptacle must be a three-pronged type (AC ground). If not, contact an electrician to install the proper receptacle. If a multi-connector box is used to connect the computer and peripherals to AC, the ground must be common to all units.

If necessary, consult your Commodore dealer or an experienced radio-television technician for additional suggestions. You may find the following FCC booklet helpful: "How to Identify and Resolve Radio-TV Interference Problems." The booklet is available from the U.S. Government Printing Office, Washington, D.C. 20402, stock no. 004-000-00345-4.

1581
Disk Drive
User's Guide

CONTENTS

INTRODUCTION

The Commodore 1581 is a versatile 3.5″ disk drive that can be used with a variety of computers, including the COMMODORE 128™, the COMMODORE 64®, the Plus 4™, COMMODORE 16™, and VIC 20.™

Also, in addition to the convenience of 3.5″ disks, the 1581 offers the following features:

- *Standard and fast serial data transfer rates*—The 1581 automatically selects the proper data transfer rate (fast or slow) to match the operating modes available on the Commodore 128 computer.
- *Double-sided, double-density MFM data recording*—Provides more than 800K formatted storage capacity per disk (400K + per side).
- *Special high-speed burst commands*—These commands, used for machine language programs, transfer data several times faster than the standard or fast serial rates.

HOW THIS GUIDE IS ORGANIZED

This guide is divided into two main parts and seven appendices, as described below:

PART ONE: BASIC OPERATING INFORMATION—includes all the information needed by novices and advanced users to set up and begin using the 1581 disk drive. PART ONE is subdivided into three chapters:

- Chapter 1, *Unpacking, Setting Up and Using the 1581,* tells you how to use disk software programs that you buy. These prewritten programs help you perform a variety of activities in fields such as business, education, finance, science, and recreation. If you're interested only in loading and running prepackaged disk programs, you need read no further than this chapter. If you are also interested in saving, loading, and running your own programs, you will want to read the remainder of the guide.
- Chapter 2, *Basic 2.0 Commands,* describes the use of the BASIC 2.0 disk commands with the Commodore 64 and Commodore 128 computers.
- Chapter 3, *Basic 7.0 Commands,* describes the use of the BASIC 7.0 disk commands with the Commodore 128.

PART TWO: ADVANCED OPERATION AND PROGRAMMING—is primarily intended for users familiar with computer programming. PART TWO is subdivided into six chapters:

- Chapter 4, *Sequential Data Files,* discusses the concept of data files, defines sequential data files, and describes how sequential data files are created and used on disk.
- Chapter 5, *Relative Data Files,* defines the differences between sequential and relative data files, and describes how relative data files are created and used on disk.
- Chapter 6, *Direct Access Commands,* describes direct access disk commands as a tool for advanced users and illustrates their use.
- Chapter 7, *Internal Disk Commands,* centers on internal disk commands. Before using these advanced commands, you should know how to program a 6502 chip in machine language.
- Chapter 8, *Machine Language Programs,* provides a list of disk-related kernal ROM subroutines and gives a practical example of their use in a program.
- Chapter 9, *Burst Commands,* gives information on high-speed burst commands.
- Chapter 10, *1581 Internal Operations,* describes how the 1581 operates internally.

APPENDICES A THROUGH F—provide various reference information; for example, Appendix A tells you how to set the device number through use of two switches on the back of the drive.

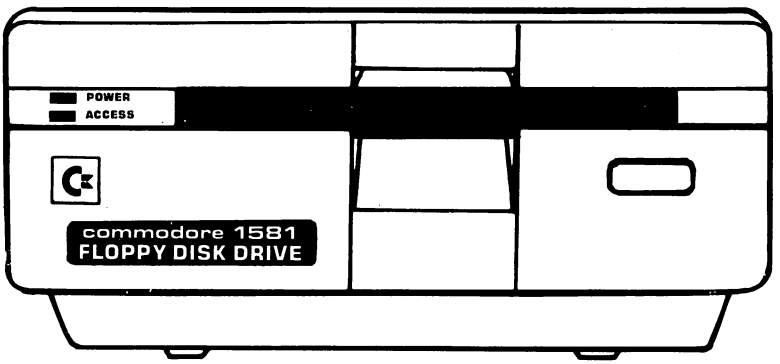
PART ONE
BASIC OPERATING INFORMATION

CHAPTER 1
HOW TO UNPACK, SET UP AND BEGIN USING THE 1581

STEP-BY-STEP INSTRUCTIONS

1. Inspect the shipping carton for damage.
   If you find any damage to the shipping carton and suspect that the disk drive may have been affected, contact your dealer.
2. Check the contents of the shipping carton.
   Packed with the 1581 and this book, you should find the following: electrical power supply, interface cable, Test/Demo diskette, and a warranty card to be filled out and returned to Commodore.
3. Remove the shipping spacer from the disk drive.
   The spacer is there to protect the inside of the drive during shipping. To remove it, push the button on the front of the drive (see Figure 1) and pull out the spacer.

Figure 1. Front of 1581 Disk Drive

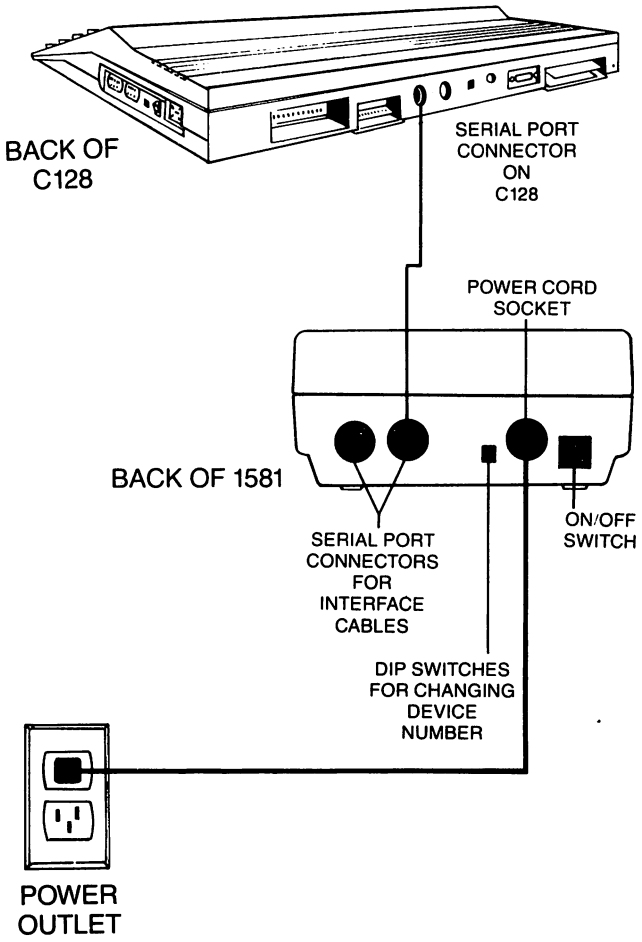


THE PRODUCT DOES NOT NECESSARILY RESEMBLE THE PICTURE INSIDE THE USER'S MANUAL

4. Connect the power cord.

Check the ON/OFF switch on the back of the drive (see Figure 2) and make sure it's OFF. Connect the power supply where indicated in Figure 2. Plug the other end into an electrical outlet. Don't turn the power on yet.

Figure 2. Connection of Power Cord and Interface Cables to 1581



5. Connect the interface cable.

   Make sure your computer and any other peripherals are OFF. Plug either end of the interface cable into either serial port on the back of the drive (see Figure 2). Plug the other end of the cable into the back of the computer. If you have another peripheral (printer or extra drive), plug its interface cable into the remaining serial port on the drive.

6. Turn ON the power.

   With everything hooked up and the drive empty, you can turn on the power to the peripherals in any order, but turn on the power to the computer last. When everything is on, the drive goes through a self test. If all is well, the green light will flash once and the red power-on light will glow continuously. If the red light continues to flash, there may be a problem. In that case, refer to the Trouble-shooting Guide.

TROUBLESHOOTING GUIDE

Problem	Possible Cause	Solution
Red power-on indicator not lit	Power not ON	Make sure ON/OFF switch is ON
	Power cable not plugged in	Check both ends of power cable to be sure they are fully inserted
	Power off to wall outlet	Replace fuse or reset circuit breaker in house
Red light flashing	Drive failing its self test	Turn the system off for a moment then try again. If the light still flashes, turn the drive off and on again with the interface cable disconnected. If the problem persists, contact your dealer. If unplugging the interface cable made a difference, make sure the cable is properly connected. If that doesn't work, the problem is probably in the cable itself or somewhere else in the system

TROUBLESHOOTING GUIDE (Cont.)

Programs won't load and the computer says "DEVICE NOT PRESENT ERROR"	Interface cable not well connected or drive not ON	Be sure the cable is properly connected and the drive is ON
	Switches on back of drive may not be set for correct device number	Check Appendix A for correct setting to match LOAD command
Programs won't load, but the computer and disk drive give no error message	Another part of the system may be interfering	Unplug all other machines on the computer. If that cures it, plug them in one at a time. The one just added when the trouble repeats is most likely the problem
		Trying to load a machine language program into BASIC space will cause this problem
Programs won't load and red light flashes	Disk error	Check the error channel to determine the error, then follow the advice in Appendix B to correct it. The error channel is explained in Chapters 2 and 3
(Be sure to spell program names correctly and include the exact punctuation when loading the programs)		
Your programs load OK, but commercial programs and those from other 1581s don't	Either the diskette is faulty, or your disk drive is misaligned	Try another copy of the program. If several programs from several sources fail to load, have your dealer align your disk drive
Your programs that used to load, won't anymore, but programs saved on newly-formatted diskettes will	Older diskettes have been damaged	Recopy from backups
	The disk drive has gone out of alignment	Have your dealer align your disk drive

TIPS FOR MAINTENANCE AND CARE

1. Keep the drive well ventilated.
   A couple of inches of space to allow air circulation on all sides will prevent heat from building up inside the drive.
2. The 1581 should be cleaned once a year in normal use.
   Several items are likely to need attention: the two read/write heads may need cleaning (with 91% isopropyl alcohol on a cotton swab). The rails along which the head moves may need lubrication (with a special molybdenum lubricant, not oil), and the write protect sensor may need to be dusted. Since these chores require special materials or parts, it is best to leave the work to an authorized Commodore service center. If you want to do the work yourself, ask your dealer for the appropriate materials. IMPORTANT: Home repair of the 1581 will void your warranty.
3. Use good quality diskettes.
   Badly-made diskettes can cause increased wear on the drive's read/write head. If you're using a diskette that is unusually noisy, it could be causing added wear and should be replaced.
4. Keep diskettes (and disk drive) away from magnets.
   That includes the electromagnets in telephones, televisions, desk lamps, and calculator cords. Keep smoke, moisture, dust, and food off the diskettes.
5. Remove a diskette before turning the drive off.
   If you don't, you might lose part or all the data on the diskette.
6. Don't remove a diskette from the drive while the green light is glowing *and* the drive motor is turning.
   If you remove the diskette then, you might lose the information currently being written to the diskette.

INSERTING A DISKETTE

Grasp the diskette by the side opposite the metal shutter. Hold it with the label up and the write-protect notch down and to the left (See Figüre 3). Insert the diskette by pushing it straight into the drive with the access slot going in first and the label last. Be sure the diskette goes in until it drops into place; you shouldn't have to force it.

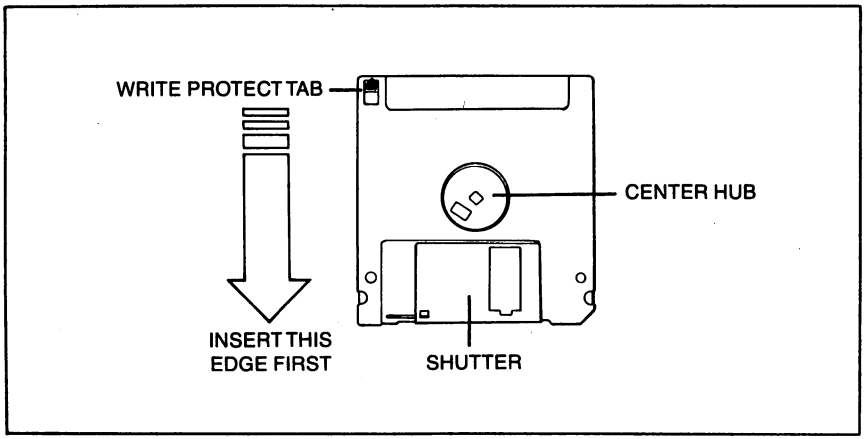


Figure 3. Inserting a Diskette

When the write/protect notch is open, the contents of the diskette cannot be altered or added to. That prevents accidental erasing of information you want to preserve.

Blank diskettes may not have a label on them when you purchase them.

USING PRE-PROGRAMMED (SOFTWARE) DISKETTES

Your software user's guide should list the procedure for loading the program into your computer. Nevertheless, we've included the following procedure as a general guide. You'll also use this procedure to load programs or files from your own diskettes. For purposes of demonstration, use the Test/Demo diskette included with the disk drive.

1. Turn on system.
2. Insert diskette.
3. If you are using a VIC 20, Commodore 64, or a Commodore 128 computer in C64 mode, type: LOAD "HOW TO USE",8
   If you are using a Plus/4 or Commodore 128 in C128 mode, type: DLOAD "HOW TO USE"
4. Press the RETURN key.
5. The following will then appear on the screen:

```
SEARCHING FOR 0:HOW TO USE
LOADING

READY

■
```

6. Type: RUN
7. Press the RETURN key.

To load a different program or file, simply substitute its name in place of HOW TO USE inside the quotation marks.

NOTE

The HOW TO USE program is the key to the Test/Demo diskette. When you LOAD and RUN it, it provides instructions for using the rest of the programs on the diskette. To find out what programs are on your Test/Demo diskette, refer to the section entitled "DIRECTORIES" later in this chapter.

If a program doesn't load or run properly using the above method, it may be that it is a machine language program. But unless you'll be doing advanced programming, you need not know anything about machine language. A program's user's guide should tell you if it is written in machine language. If it is, or if you are having trouble loading a particular program, simply add a ,1 (comma and number 1) at the end of the command.

NOTE

Throughout this manual, when the format for a command is given, it will follow a particular style. Anything that is capitalized must be typed in exactly as it is shown (these commands are listed in capital letters for style purposes, DO NOT use the SHIFT key when entering these commands). Anything in lower case is more or less a definition of what belongs there. Anything in brackets is optional.

For instance, in the format for the HEADER command given on the following page, the word HEADER, the capital I in Iid, the capital D in Ddrive#, and the capital U in Udevice# must all be typed in as is (Ddrive# and Udevice# are optional).

On the other hand, diskette name tells you that you must enter a name for the diskette, but it is up to you to decide what that name will be. Also, the id in Iid is left to your discretion, as is the device# in Udevice#. The drive# in Ddrive# is always 0 on the 1581, but could be 0 or 1 on a dual disk drive. Be aware, however, that there are certain limits placed on what you can use. In each case, those limits are explained immediately following the format (for instance, the diskette name cannot be more than sixteen characters and the device# is usually 8).

Also be sure to type in all punctuation exactly where and how it is shown in the format.

Finally, press the RETURN key at the end of each command.

HOW TO PREPARE A NEW DISKETTE

A diskette needs a pattern of circular magnetic tracks in order for the drive's read/write head to find things on it. This pattern is not on your diskettes when you buy them, but you can use the HEADER command or the NEW command to add it to a diskette. That is known as formatting the disk. This is the command to use with the C128 in C128 mode or Plus/4:

HEADER "diskette name",Iid,Ddrive#[,Udevice#]

Where:
"diskette name" is any desired name for the diskette, up to 16 charac-

ters long (including spaces). "id" can be any two characters as long as they don't form a BASIC keyword (such as IF or ON) either on their own or with the capital I before them. "drive#" is 0 . "device#" is 8, unless you have changed it as per instructions in Appendix A (the 1581 assumes 8 even if you don't type it in).

The command for the C64, VIC 20, or C128 in C64 mode is this:

```
OPEN 15,device#,15,"NEWdrive#:diskette name,id"
CLOSE 15
```

The device#, drive#, diskette name, and id are the same as described above.

The OPEN command is explained in the next chapter. For now, just copy it as is.

NOTE

FOR ADVANCED USERS

If you want to use variables for the diskette name or id, the format is as follows:

C128, Plus/4: HEADER (A$),I(B$),D0

C64:
```
OPEN 15,8,15:PRINT#15,"N0:"+A$+","+B$:
  CLOSE15
```

Where:

A$ contains the diskette name (16 character limit)
B$ contains the id (2 characters long)

After you format a particular diskette, you can reformat it at any time. You can change its name and erase its files faster by omitting the id number in the HEADER command.

DISKETTE DIRECTORY

A directory is a list of the files on a diskette. To view the directory on the C128 or Plus/4, type the word DIRECTORY on a blank line and press the RETURN key or simply press the F3 key on the C128. That doesn't erase anything in memory, so you can call up a directory

anytime—even from within a program. The C64 directory command, LOAD "$",8 (press RETURN) LIST (press RETURN), does erase what's in memory.

If a directory doesn't all fit on the screen, it will scroll up until it reaches the last line. If you want to pause, stop, or slow down the scrolling, refer to your particular computer's user's manual for instructions as to which keys to use.

To get an idea of what a directory looks like, load the directory from the Test/Demo diskette.

The 0 on the left-hand side of the top line is the drive number of the 1581. The diskette name is next, followed by the diskette id—both of which are determined when the diskette is formatted.

The 3D at the end of the top line means the 1581 uses Version 3D of Commodore's disk operating system (DOS).

Each of the remaining lines provides three pieces of information about the files on the diskette. At the left end of each line is the size of the file in blocks of 254 characters. Four blocks are equivalent to almost 1K of memory inside the computer. The middle of the line contains the name of the file enclosed in quotation marks. All characters within the quotation marks are part of the filename. The right side of each line contains a three-letter abbreviation of the file type. The types of files are described in later chapters.

The bottom line of a directory shows how many blocks are available for use. This number ranges from 3160 on a newly formatted diskette to 0 on one that is completely full.

SELECTIVE DIRECTORIES

By altering the directory LOAD command, you can create a subdirectory that lists a single selected type of file. For example, you could request a list of all sequential data files (Chapter 4), or one of all the relative data files (Chapter 5). The format for this command is:

LOAD"$0:pattern=filetype",8 (for the C64)

where pattern specifies a particular group of files, and filetype is the one-letter abbreviation for the types of files listed below:

P = Program
S = Sequential
R = Relative
U = User

The command for the C128 and Plus/4 is this:

DIRECTORY"pattern = filetype"

Some examples:

LOAD"$0:* = R",8 and DIRECTORY"* = R" display all relative files.

LOAD"$0:Z* = R",8 and DIRECTORY"Z* = R" display a sub-directory consisting of all relative files that start with the letter Z (the asterisk (*) is explained in the section entitled "Pattern Matching."

PRINTING A DIRECTORY

To printout a directory, use the following:

```
LOAD"$",8
OPEN4,4:CMD4:LIST
PRINT#4:CLOSE4
```

PATTERN MATCHING

You can use special pattern-matching characters to load a program from a partial name or to provide the selective directories described earlier.

The two characters used in pattern matching are the asterisk (*) and the question mark (?). They act something like a wild card in a game of cards. The difference between the two is that the asterisk makes all characters in and beyond its position wild, while the question mark makes only its own position wild. Here are some examples and their results:

LOAD "A*",8 loads the first file on disk that begins with an A, regardless of what follows

DLOAD"SM?TH" loads the first file that starts with SM, ends with TH, and one other character between

DIRECTORY"Q*" loads a directory of files whose names begin with Q

LOAD"*",8 is a special case. When an asterisk is used alone as a name, it matches the last file used (on the C64 and C128 in C64 mode).

LOAD "0:*",8 loads the first file on the diskette (C64 and C128 in C64 mode).

DLOAD "*" loads the first file on the diskette (Plus/4 and C128 in C128 mode).

SPLAT FILES

One indicator you may occasionally notice on a directory line, after you begin saving programs and files, is an asterisk appearing just before the file type of a file that is 0 blocks long. This indicates the file was not properly closed after it was created, and that it should not be relied upon. These "splat" files normally need to be erased from the diskette and rewritten. However, do not use the SCRATCH command to get rid of them. They can only be safely erased by the VALIDATE or COLLECT commands. One of these should normally be used whenever a splat file is noticed on a diskette. All of these commands are described in the following chapters.

There are two exceptions to the above warning: one is that VALIDATE and COLLECT cannot be used on some diskettes that include direct access (random) files (Chapter 6). The other is that if the information in the splat file was crucial and can't be replaced, there is a way to rescue whatever part of the file was properly written. This option is described in the next chapter.

CHAPTER 2
BASIC 2.0 COMMANDS

This chapter describes the disk commands used with the VIC 20, Commodore 64 or the Commodore 128 computer in C64 mode. These are Basic 2.0 commands.

You send command data to the drive through something called the command channel. The first step is to open the channel with the following command:

```
OPEN15,8,15
```

The first 15 is a file number or channel number. Although it could be any number from 1 to 255, we'll use 15 because it is used to match the secondary address of 15, which is the address of the command channel. The middle number is the primary address, better known as the device number. It is usually 8, unless you change it (see Appendix A).

Once the channel has been opened, use the PRINT# command to send information to the disk drive and the INPUT# command to receive information from the drive. You must close the channel with the CLOSE15 command.

The following examples show the use of the command channel to NEW an unformatted disk:

```
OPEN15,8,15
PRINT#15,"NEWdrive#:diskname,id"
CLOSE15
```

You can combine the first two statements and abbreviate the NEW command like this:

```
OPEN15,8,15,"Ndrive# :diskname,id"
```

If the command channel is already open, you must use the following format (trying to open a channel that is already open results in a "FILE OPEN" error):

```
PRINT#15,"Ndrive#:diskname,id"
```

ERROR CHECKING

In Basic 2.0, when the red drive light flashes, you must write a small program to find out what the error is. This causes you to lose any program variables already in memory. The following is the error check program:

```
10 OPEN15,8,15
20 INPUT#15,EN,EM$,ET,ES
30 PRINT EN, EM$,ET,ES
40 CLOSE15
```

This little program reads the error channel into four BASIC variables (described below), and prints the results on the screen. A message is displayed whether there is an error or not, but if there was an error, the program clears it from disk memory and stops the error light from blinking.

Once the message is on the screen, you can look it up in Appendix B to see what it means, and what to do about it.

For those of you who are writing programs, the following is a small error-checking subroutine you can include in your programs:

```
59980 REM READ ERROR CHANNEL
59990 INPUT#15,EN,EM$,ET,ES
60000 IF EN>1 THEN PRINT EN,EM$,ET,ES:STOP
60010 RETURN
```

This assumes file 15 was opened earlier in the program, and that it will be closed at the end of the program.

The subroutine reads the error channel and puts the results into the named variables—EN (Error Number), EM$ (Error Message), ET (Error Track), and ES (Error Sector). Of the four, only EM$ has to be a string. You could choose other variable names, although these have become traditional for this use.

Two error numbers are harmless—0 means everything is OK, and 1 tells how many files were erased by a SCRATCH command (described later in this chapter). If the error status is anything else, line 60000 prints the error message and halts the program.

Because this is a subroutine, you access it with the BASIC GOSUB command, either in immediate mode or from a program. The RETURN statement in line 60010 will jump back to immediate mode or the next statement in your program, whichever is appropriate.

BASIC HINTS

1. It is best to open file 15 once at the very start of a program, and only close it at the end of the program, after all other files have already been closed. By opening it once at the start, the file is open whenever needed for disk commands elsewhere in the program.

2. If BASIC halts with an error when you have files open, BASIC aborts them without closing them properly on the disk. To close them properly on the disk, you must type:

CLOSE 15:OPEN 15,8,15,"I":CLOSE 15

This opens the command channel and immediately closes it, along with all other disk files. Failure to close a disk file properly both in BASIC and on the disk may result in losing the entire file.

3. One disk error message is not always an error. Error 73, "COPYRIGHT CBM DOS V10 1581" will appear if you read the disk error channel before sending any disk commands when you turn on your computer. This is a handy way to check which version of DOS you are using. However, if this message appears later, after other disk commands, it means there is a mismatch between the DOS used to format your diskette and the DOS in your drive. DOS is Disk Operating System.

4. To reset drive, type: OPEN 15,8,15,"UJ":CLOSE 15.

SAVE

The SAVE command preserves a program or file on a formatted diskette for later use.

FORMAT FOR THE SAVE COMMAND

SAVE "drive #:file name",device #

where "file name" is any string expression of up to 16 characters, preceded by the drive number and a colon, and followed by the device number of the disk, normally 8.

However, the SAVE command will not work in copying programs that are not in the BASIC text area, such as "DOS 5.1" for the C64. To copy it and similar machine-language programs, you will need a machine-language monitor program, such as the one resident in the C128.

FORMAT FOR A MONITOR SAVE

S "drive #:file name",device #,starting address,ending address + 1

where "drive #:" is the drive number, 0 on the 1581; "file name" is any valid file name up to 14 characters long (leaving two for the drive number and colon); "device #" is a two digit device number, normally 08 (the leading 0 is required); and the addresses to be saved are given in Hexadecimal but without a leading dollar sign ($). Note the ending address listed must be one location beyond the last location to be saved.

EXAMPLE:

Here is the required syntax to SAVE a copy of "DOS 5.1"

S "0:DOS 5.1",08,CC00,D000

SAVE WITH REPLACE

If a file already exists, it can't be saved again with the same name because the disk drive only allows one copy of any given file name per diskette. It is possible to get around this problem using the RENAME and SCRATCH commands described later. However, if all you wish to do is replace a program or data file with a revised version, another command is more convenient. Known as SAVE-WITH-REPLACE, or @SAVE, this option tells the disk drive to replace any file it finds in the diskette directory with the same name, substituting the new file for the old version.

FORMAT FOR SAVE WITH REPLACE:

SAVE"@Drive #:file name", device #

where all the parameters are as usual except for adding a leading "at" sign (@.) The "drive #:" is required here.

EXAMPLE:

SAVE"@0:REVISED PROGRAM",8

The actual procedure is that the new version is saved completely, then the old version is erased. Because it works this way, there is little

danger a disaster such as losing power midway through the process would destroy both the old and new copies of the file. Nothing happens to the old copy until after the new copy is saved properly.

VERIFY

The VERIFY command can be used to make certain that a program file was properly saved to disk. It works much like the LOAD command, except that it only compares each character in the program against the equivalent character in the computer's memory, instead of actually being copied into memory.

If the disk copy of the program differs even a tiny bit from the copy in memory, "VERIFY ERROR" will be displayed, to tell you that the copies differ. This doesn't mean either copy is bad, but if they were supposed to be identical, there is a problem.

Naturally, there's no point in trying to VERIFY a disk copy of a program after the original is no longer in memory. With nothing to compare to, an apparent error will always be announced, even though the disk copy is automatically verified as it is written to the diskette.

FORMAT FOR THE VERIFY COMMAND:

VERIFY "drive#:pattern",device#,relocate flag

where "drive#:" is an optional drive number, "pattern" is any string expression that evaluates to a file name, with or without pattern-matching characters, and "device#" is the disk device number, normally 8. If the relocate flag is present and equals 1, the file will be verified where originally saved, rather than relocated into the BASIC text area.

A useful alternate form of the command is:

VERIFY"*",device #

It verifies the last files used without having to type its name or drive number. However, it won't work properly after SAVE-WITH-REPLACE, because the last file used was the one deleted, and the drive will try to compare the deleted file to the program in memory. No harm will result, but "VERIFY ERROR" will always be announced. To use VERIFY after @SAVE, include at least part of the file name that is to be verified in the pattern.

One other note about VERIFY—when you VERIFY a relocated BASIC file, an error will nearly always be announced, due to changes in the link pointers of BASIC programs made during relocation. It is best to VERIFY files saved from the same type of machine, and identical memory size. For example, a BASIC program saved from a Plus/4 can't be verified easily with a C64, even when the program would work fine on both machines. This shouldn't matter, as the only time you'll be verifying files on machines other than the one which wrote them is when you are comparing two disk files to see if they are the same. This is done by loading one and verifying against the other, and can only be done on the same machine and memory size as the one on which the files were first created.

SCRATCH

The SCRATCH command allows you to erase unwanted files and free the space they occupied for use by other files. It can be used to erase either a single file or several files at once via pattern-matching.

FORMAT FOR THE SCRATCH COMMAND:

```
PRINT#15,"SCRATCH0:pattern"
```

or abbreviate it as:

```
PRINT#15,"S0:pattern"
```

"pattern" can be any file name or combination of characters and wildcard characters. As usual, it is assumed the command channel has already been opened as file 15. Although not absolutely necessary, it is best to include the drive number in SCRATCH commands.

If you check the error channel after a SCRATCH command, the value for ET (error track) will tell you how many files were scratched. For example, if your diskette contains program files named "TEST," "TRAIN," "TRUCK," and "TAIL," you may SCRATCH all four, along with any other files beginning with the letter "T," by using the command:

```
PRINT#15,'S0:T*'
```

Then, to prove they are gone, you can type:

```
GOSUB 59990
```

to call the error checking subroutine given earlier in this chapter. If the four listed were the only files beginning with "T", you will see:

```
01,FILES SCRATCHED,04,00

READY.
```

The "04" tells you 4 files were scratched.

MORE ABOUT SCRATCH

SCRATCH is a powerful command and should be used with caution to be sure you delete only the files you really want erased. When using it with a pattern, we suggest you first use the same pattern in a DIRECTORY command, to be sure exactly which files will be deleted. That way you'll have no unpleasant surprises when you use the same pattern in the SCRATCH command.

If you accidentally SCRATCH a file you shouldn't have, there is still a chance of saving it by using the "Unscratch" program on your Test/Demo diskette.

More about Splats

Never scratch a splat file. These are files that show up in a directory listing with an asterisk (*) just before the file type for an entry. The asterisk (or splat) means that file was never properly closed, and thus there is no valid chain of sector links for the SCRATCH command to follow in erasing the file.

If you SCRATCH such a file, odds are you will improperly free up sectors that are still needed by other programs or files and cause permanent damage to those later when you add more files to the diskette. If you find a splat file, or if you discover too late that you have scratched such a file, immediately validate the diskette using the VALIDATE command described later in this chapter. If you have added any files to the diskette since scratching the splat file, it is best to immediately copy the entire diskette onto another fresh diskette, but do this with a copy program rather than with a backup program. Otherwise, the same problem will be recreated on the new diskette. When the new copy is done, compare the number of blocks free in its directory to the number free on the original diskette. If the numbers match, no damage has been done. If not, very likely at least one file on the diskette has been corrupted, and all should be checked immediately.

Locked Files

Occasionally, a diskette will contain a locked file; one which cannot be erased with the SCRATCH command. Such files may be recognized by the "<" character which immediately follows the file type in their directory entry. If you wish to erase a locked file, you will have to use a sector editor program to clear bit 6 of the file-type byte in the directory entry on the diskette. Conversely, to lock a file, you would set bit 6 of the same byte.

RENAME

The RENAME command allows you to alter the name of a program or other file in the diskette directory. Since only the directory is affected, RENAME works very quickly.

FORMAT FOR RENAME COMMAND:

PRINT#15,"RENAME0:new name=old name"

or it may be abbreviated as:

PRINT#15,"R0:new name=old name"

where "new name" is the name you want the file to have, and "old name" is the name it has now. "new name" may be any valid file name, up to 16 characters in length. It is assumed you have already opened file 15 to the command channel.

One caution—be sure the file you are renaming has been properly closed before you rename it.

EXAMPLES:

Just before saving a new copy of a "calendar" program, you might type:

PRINT#15,"R0:CALENDAR/BACKUP=CALENDAR"

Or to move a program called "BOOT," currently the first program on a diskette to someplace else in the directory, you might type:

PRINT#15,"R0:TEMP=BOOT"

followed by a COPY command (described later), which turns "TEMP" into a new copy of "BOOT," and finishing with a SCRATCH command to get rid of the original copy of "BOOT."

RENAMING AND SCRATCHING TROUBLESOME FILES

Eventually, you may run across a file which has an odd filename, such as a comma by itself (",") or one that includes a Shifted Space (a Shifted Space looks the same as a regular space, but if a file with a space in its name won't load properly and all else is correct, it's probably a Shifted Space). Or perhaps you will find one that includes nonprinting characters. Any of these can be troublesome. Comma files, for instance, are an exception to the rule that no two files can have the same name. Since it shouldn't be possible to make a file whose name is only a comma, the disk never expects you to do it again.

Files with a Shifted Space in their name can also be troublesome, because the disk interprets the Shifted Space as signaling the end of the file name, and prints whatever follows after the quotation mark that marks the end of a name in the directory. This technique can be useful by allowing you to have a long file name, and making the disk recognize a small part of it as being the same as the whole thing without using pattern-matching characters.

In any case, if you have a troublesome filename, you can use the CHR$() function to specify troublesome characters without typing them directly. This may allow you to build them into a RENAME command. If this fails, you may also use the pattern-matching characters in a SCRATCH command. This gives you a way to specify the name without using the troublesome characters at all, but also means loss of your file.

For example, if you have managed to create a file named ""MOVIES", with an extra quotation mark at the front of the file name, you can rename it to "MOVIES" using the CHR$() equivalent of a quotation mark in the RENAME command:

```
PRINT#15,"R0:MOVIES="+CHR$(34)+"MOVIES"
```

The CHR$(34) forces a quotation mark into the command string without upsetting BASIC. The procedure for a file name that includes a Shifted Space is similar, but uses CHR$(160).

In cases where even this doesn't work, for example, if your diskette contains a comma file, (one named ",") you can get rid of it this way:

```
PRINT#15,"S0:?"
```

This example deletes all files with one character names.

Depending on the exact problem, you may have to be very creative in choosing pattern-matching characters that will affect only the desired file, and may have to rename other files first to keep them from being scratched.

In some cases, it may be easier to copy desired files to a different diskette and leave the troublesome files behind.

COPY

The COPY command allows you to make a spare copy of any program or file on a diskette. On a single drive like the 1581, the copy must be on the same diskette, which means it must be given a different name from the file copied. It's also used to combine up to four sequential data files (linking the files one to another, end to end in a chain). Files are linked in the order in which they appear in the command. The source files and other files on the diskette are not changed. Files must be closed before they are copied or linked.

FORMAT FOR THE COPY COMMAND

PRINT#15,"COPYdrive #:new file=old file"

EXAMPLES:

PRINT#15,"COPY0:BACKUP=ORIGINAL"

or abbreviated as

PRINT#15,"Cdrive #:new file=old file"

PRINT#15,"C0:BACKUP=ORIGINAL"

where "drive #" is the drive number "new file" is the copy and "old file" is the original.

FORMAT FOR THE COMBINE OPTION

PRINT#15,"Cdrive #:new file=file 1,file 2,file 3, file 4"

where "drive #" is always 0,

NOTE

The length of a command string (command and filenames) is limited to 41 characters.

EXAMPLES:

After renaming a file named "BOOT" to "TEMP" in the last section's example, you can use the COPY command to make a spare copy of the program elsewhere on the diskette, under the original name:

PRINT#15,"C0:BOOT=TEMP"

After creating several small sequential files that fit easily in memory along with a program we are using, you can use the concatenate option to combine them in a master file, even if the result is too big to fit in memory. (Be sure it will fit in remaining space on the diskette—it will be as big as the sum of the sizes of the files in it.)

PRINT#15,"C0:A-Z=A-G,H-M,N-Z"

EXAMPLES:

After renaming a file named "BOOT" to "TEMP" in the last section's example, you can use the COPY command to make a spare copy of the program elsewhere on the diskette, under the original name:

PRINT#15,"C0:BOOT=TEMP"

After creating several small sequential files that fit easily in memory along with a program we are using, you can use the concatenate option to combine them in a master file, even if the result is too big to fit in memory. (Be sure it will fit in remaining space on the diskette—it will be as big as the sum of the sizes of the files in it.)

PRINT#15,"C0:A-Z=A-G,H-M,N-Z"

VALIDATE

The VALIDATE command recalculates the Block Availability Map (BAM) of the current diskette, allocating only those sectors still being used by valid, properly-closed files and programs. All other sectors (blocks) are left unallocated and free for re-use, and all improperly closed files are automatically scratched. This brief description of its workings doesn't indicate either the power or the danger of the

VALIDATE command. Its power is in restoring to good health many diskettes whose directories or block availability maps have become muddled. Any time the blocks used by the files on a diskette plus the blocks shown as free don't add up to the 3160 available on a fresh diskette, VALIDATE is needed, with one exception below. Similarly, any time a diskette contains an improperly-closed file (splat file), indicated by an asterisk (*) next to its file type in the directory, that diskette needs to be validated. In fact, but for the one exception, it is a good idea to VALIDATE diskettes whenever you are the least bit concerned about their integrity.

The exception is diskettes containing direct access files, as described in Chapter 6. Most direct access (random) files do not allocate their sectors in a way the VALIDATE command can recognize. Thus, using VALIDATE on such a diskette may result in un-allocating all direct access files, with loss of all their contents when other files are added. Unless specifically instructed otherwise, never use VALIDATE on a diskette containing direct access files.

FORMAT FOR THE VALIDATE COMMAND

```
PRINT#15,"VALIDATE0"
```

or abbreviated as

```
PRINT#15,"V0"
```

where "0" is the drive number. As usual, it is assumed file 15 has been opened to the command channel and will be closed after the command has been executed.

INITIALIZE

When a diskette is inserted into the drive, its directory is automatically re-read into a disk buffer. You would use the command only if that information became unreliable.

FORMAT FOR THE INITIALIZE COMMAND

```
PRINT#15,"INITIALIZEdrive#"
```

or it may be abbreviated to

```
PRINT#15,"Idrive#"
```

CHAPTER 3
BASIC 7.0 COMMANDS

This chapter describes the disk commands used with the Commodore 128 computer (in C128 mode). This is BASIC 7.0, which includes BASIC 2.0, BASIC 3.5, and BASIC 4.0 commands, all of which can be used.

ERROR CHECKING

When the drive light (red light) flashes, you must use the following command to find out what the error is:

```
PRINT  DS$
```

A message is displayed whether there is an error or not. If there was an error, this command clears it from disk memory and turns off the error light on the disk drive.

Once the message is on the screen, you can look it up in Appendix B to see what it means, and what to do about it.

For those of you who are writing programs, the following is a small error-checking subroutine you can include in your programs:

```
59990  REM READ ERROR CHANNEL
60000  IF DS>1 THEN PRINT DS$:STOP
60010  RETURN
```

The subroutine reads the error channel and puts the results into the reserved variables DS and DS$. They are updated automatically by BASIC.

Two error numbers are harmless—0 means everything is OK, and 1 tells how many files were erased by a SCRATCH command (described later in this chapter). If the error status is anything else, line 60000 prints the error message and halts the program.

Because this is a subroutine, you access it with the BASIC GOSUB command, either in immediate mode or from a program. The RETURN statement in line 60010 will jump back to immediate mode or the next statement in your program, whichever is appropriate.

SAVE

This command will save a program or file so you can reuse it. The diskette must be formatted before you can save it to that diskette.

FORMAT FOR THE SAVE COMMAND

DSAVE "file name" [,Ddrive#] [,Udevice#]

This command will not work in copying programs that are not written in BASIC. To copy these machine language programs, you can use the BSAVE command or the built-in Monitor S command.

FORMAT FOR THE BSAVE COMMAND

BSAVE "file name" [,Ddrive#] [,Udevice#] [Bbank#]
[,Pstarting address] [TO Pending address + 1]

where the usual options are the same and bank# is one of the 16 banks of the C128. The addresses to be saved are given in decimal. Note that the ending address must be 1 location beyond the last location to be saved.

To access a built-in monitor, type MONITOR. To exit a monitor, type X alone on a line.

FORMAT FOR A MONITOR SAVE

S"drive #:file name",device #,starting address,ending address + 1

where "drive #:" is the drive number, 0 on the 1581; "file name" is any valid file name up to 14 characters long (16 if you leave out the drive # and the colon that follows it); "device #" is a two digit device number, normally 08 on the 1581 (the leading 0 is required); and the addresses to be saved are given in Hexadecimal (base 16,) but without a leading dollar sign (for the Plus/4). On the C128, the addresses need not be in Hexidecimal. Note that the ending address listed must be 1 location beyond the last location to be saved.

SAVE WITH REPLACE

If a file already exists, it can't be saved again with the same name because the disk drive allows only one copy of any given file name per diskette. It is possible to get around this problem using the RENAME and SCRATCH commands described later in this chapter. If all you wish to do is replace a program or data file with a revised version, another command is more convenient. Known as SAVE WITH REPLACE, or @SAVE this option tells the disk drive to replace any file it finds in the diskette directory with the same name, substituting the new file for the old version.

FORMAT FOR SAVE WITH REPLACE

DSAVE "@file name" [,Ddrive#] [,Udevice#]

The actual procedure is this—the new version is saved completely, then the old version is scratched and its directory entry altered to point to the new version. Because it works this way, there is little danger a disaster such as having the power going off midway through the process would destroy both the old and new copies of the file. Nothing happens to the old copy until after the new copy is saved properly.

VERIFY

This command makes a byte-by-byte comparison of the program currently in memory against a program on diskette. This comparison includes the BASIC line links, which may be different for different types of memory configurations. What this means is that a program saved to disk on a C64 and reloaded into a C128 wouldn't verify properly because the line links point to different memory locations. If the disk copy of the program differs at all from the copy in memory, a "VERIFY ERROR" will be displayed. This doesn't mean either copy is bad, but if they were supposed to be identical, there is a problem.

FORMAT FOR THE DVERIFY COMMAND

DVERIFY "file name" [,Ddrive#] [,Udevice#]

The following version of the command verifies a file that was just saved:

DVERIFY "*"

This command won't work properly after SAVE-WITH-REPLACE, because the last file used was the one deleted and the drive will try to compare the deleted file to the program in memory. No harm will result, but "VERIFY ERROR" will always be announced. To use DVERIFY after @SAVE, include at least part of the file name that is to be verified in the pattern.

COPY

The COPY command allows you to make a spare copy of any program or file on a diskette. However, on a single drive like the 1581,

the copy must be on the same diskette, which means it must be given a different name from the file copied. The source file and other files on the diskette are not changed. Files must be closed before they can be copied or concatenated.

FORMAT FOR THE COPY COMMAND

COPY [Ddrive#,] "old file name" TO [Ddrive#,] "new file name" [,Udevice#]

Where both drive#s would be 0 if included.

CONCAT

The CONCAT command allows you to concatenate (combine) two sequential files.

FORMAT FOR THE CONCAT COMMAND

CONCAT [Ddrive#,] "add file" TO [Ddrive#,] "master file" [,Udevice#]

Where the optional drive# would be 0 in both cases. The old "master file" is deleted and replaced with a new "master file" which is the concatenation of the old "master file" and "add file".

NOTE

The length of a command string (command and filenames) is limited to 41 characters.

SCRATCH

The SCRATCH command allows you to erase unwanted programs and files from your diskettes, and free up the space they occupied for use by other files and programs. It can be used to erase either a single file, or several files at once via pattern-matching.

FORMAT FOR THE SCRATCH COMMAND

SCRATCH "pattern" [,Ddrive#] [,Udevice#]

Where, "pattern" is any valid file name or pattern-matching character.

You will be asked as a precaution:

```
ARE YOU SURE? ■
```

If you ARE sure, simply press Y and RETURN. If not, press RETURN alone or type any other answer, and the command will be canceled.

The number of files that were scratched will be automatically displayed. For example, if your diskette contains program files named "TEST," "TRAIN," "TRUCK," and "TAIL," you may scratch all four, along with any other files beginning with the letter "T," by using the command:

```
SCRATCH "T*"
```

and if the four listed were the only files beginning with "T", you will see:

```
01,FILES SCRATCHED,04,00
READY
■
```

The "04" tells you 4 files were scratched.

You can perform a SCRATCH within a program, but there will be no prompt message displayed.

MORE ABOUT SCRATCH

SCRATCH is a powerful command and should be used with caution to be sure you delete only the files you really want erased. When using it with a pattern, we suggest you first use the same pattern in a DIRECTORY command, to be sure exactly which files will be deleted. That way you'll have no unpleasant surprises when you use the same pattern in the SCRATCH command.

If you accidentally SCRATCH a file you shouldn't have, there is still a chance of saving it by using the "Unscratch" program on your Test/Demo diskette.

More about Splat Files

Never SCRATCH a splat file. These are files that show up in a directory listing with an asterisk (*) just before the file type for an entry. The asterisk (or splat) means that file was never properly closed, and thus there is no valid chain of sector links for the SCRATCH command to follow in erasing the file. If you SCRATCH such a file,

odds are you will improperly free up sectors that are still needed by other programs or files, and cause permanent damage to those other programs or files later when you add more files to the diskette.

If you find a splat file, or if you discover too late that you have scratched such a file, immediately validate the diskette using the COLLECT command described later in this chapter. If you have added any files to the diskette since scratching the splat file, it is best to immediately copy the entire diskette onto another fresh diskette, but do this with a copy program rather than with a backup program. Otherwise, the same problem will be recreated on the new diskette. When the new copy is done, compare the number of blocks free in its directory to the number free on the original diskette. If the numbers match, no damage has been done. If not, very likely at least one file on the diskette has been corrupted, and all should be checked immediately.

Locked Files

Occasionally, a diskette will contain a locked file; one which cannot be erased with the SCRATCH command. Such files may be recognized by the "<" character which immediately follows the file type in their directory entry. If you wish to erase a locked file, you will have to use a disk monitor to clear bit 6 of the file-type byte in the directory entry on the diskette. Conversely, to lock a file, you would set bit 6 of the same byte.

RENAME

The RENAME command allows you to alter the name of a program or other file in the diskette directory. Since only the directory is affected, RENAME works very quickly. If you try to RENAME a file by using a file name already in the directory, the computer will respond with a "FILE EXISTS" error. A file must be properly closed before it can be renamed.

FORMAT FOR RENAME COMMAND:

RENAME [Ddrive#,] "old name" TO [Ddrive#,] "new name" [,Udevice#]

where both drive#s, if included, would be 0

RENAMING AND SCRATCHING TROUBLESOME FILES

Eventually, you may run across a file which has a crazy filename, such as a comma by itself (",") or one that includes a Shifted Space. Or perhaps you will find one that includes nonprinting characters. Any of these can be troublesome. Comma files, for instance, are an exception to the rule that no two files can have the same name. Since it shouldn't be possible to make a file whose name is only a comma, the disk never expects you to do it again.

Files with a Shifted Space in their name can also be troublesome, because the disk interprets the Shifted Space as signaling the end of the file name, and prints whatever follows after the quotation mark that marks the end of a name in the directory. This technique can be useful by allowing you to have a long file name, and making the disk recognize a small part of it as being the same as the whole thing without using pattern-matching characters.

In any case, if you have a troublesome filename, you can use the CHR$() function to specify troublesome characters without typing them directly. This may allow you to build them into a RENAME command. If this fails, you may also use the pattern-matching characters discussed for a SCRATCH command. This gives you a way to specify the name without using the troublesome characters at all, but also means loss of your file.

For example, if you have managed to create a file named " "MOVIES", with an extra quotation mark at the front of the file name, you can rename it to "MOVIES" using the CHR$() equivalent of a quotation mark in the RENAME command:

EXAMPLE:

```
RENAME(CHR$(34)+"MOVIES") TO "MOVIES"
```

The CHR$(34) forces a quotation mark into the command string without upsetting BASIC. The procedure for a file name that includes a Shifted Space is similar, but uses CHR$(160).

In cases where even this doesn't work, for example, if your diskette contains a comma file, (one named ",") you can get rid of it this way:

```
SCRATCH"?"
```

This example deletes all files with one-character names.

Depending on the exact problem, you may have to be very creative in choosing pattern-matching characters that will affect only the desired file, and may have to rename other files first to keep them from being scratched.

In some cases, it may be easier to copy desired files to a different diskette and leave the troublesome files behind.

COLLECT

The COLLECT command recalculates the Block Availability Map (BAM) of the current diskette, allocating only those sectors still being used by valid, properly closed files and programs. All other sectors (blocks) are left unallocated and free for reuse, and all improperly closed files are automatically scratched. However, this brief description of COLLECT doesn't indicate either the power or the danger of the command. Its power is in restoring to good health many diskettes whose directories or Block Availability Maps have become muddled. Any time the blocks used by the files on a diskette plus the blocks shown as free don't add up to the 3160 available on a fresh diskette, COLLECT is needed (with one exception below). Similarly, any time a diskette contains an improperly closed file (splat file), indicated by an asterisk (*) next to its file type in the directory, that diskette needs to be collected. In fact, but for the one exception below, it is a good idea to COLLECT diskettes whenever you are concerned about their integrity. Just note the number of blocks free in the diskette's directory before and after using COLLECT. If the totals differ, there was indeed a problem, and the diskette should probably be copied onto a fresh diskette file-by-file, using the COPY command described in the previous section, rather than using a backup command or program.

The exception is diskettes containing direct access files, as described in Chapter 6. Most direct access (random) files do not allocate their sectors in a way COLLECT can recognize. Thus, collecting such a diskette may result in unallocating all direct access files, with loss of all their contents when other files are added. Unless specifically instructed otherwise, never collect a diskette containing direct access files. (Note: these are not the same as the relative files described in Chapter 5. COLLECT may be used on relative files without difficulty.)

FORMAT FOR THE COLLECT COMMAND

COLLECT [Ddrive#] [,Udevice#]

DCLEAR

One command that should not often be needed on the 1581, but is still of occasional value is DCLEAR. On the 1581, and nearly all other Commodore drives, this function is performed automatically, whenever a new diskette is inserted.

The result of an DCLEAR, whether forced by a command, or done automatically by the disk, is a re-reading of the current diskette's BAM into a disk buffer. This information must always be correct in order for the disk to store new files properly. However, since the chore is handled automatically, the only time you'd need to use the command is if something happened to make the information in the drive buffers unreliable.

FORMAT FOR THE DCLEAR COMMAND

PRINT#15,"DCLEARdrive #"

EXAMPLE:

PRINT#15,"DCLEAR 0"

or it may be abbreviated to

PRINT#15,"Idrive #"

PRINT#15,"I0"

where the command channel is assumed to be opened by file 15, and "drive #" is 0.

PART TWO
ADVANCED OPERATION AND PROGRAMMING

CHAPTER 4
SEQUENTIAL DATA FILES

A file on a diskette is just like a file cabinet in your office—an organized place to put things. Nearly everything you put on a diskette goes in one kind of file or another. So far all you've used are program files, but there are others. In this chapter you'll learn about sequential data files.

The primary purpose of a data file is to store the contents of program variables, so they won't be lost when the program ends. A sequential data file is one in which the contents of the variables are stored "in sequence," one right after another. You may already be familiar with sequential files from using a DATASSETTE™, because sequential files on diskette are just like the data files used on cassettes. Whether on cassette or diskette, sequential files must be read from beginning to end.

When sequential files are created, information (data) is transferred byte-by-byte, through a buffer, onto the magnetic media. Once in the disk drive, program files, sequential data files, and user files all work sequentially. Even the directory acts like a sequential file.

To use sequential files properly, we will learn some more BASIC words in the next few pages. Then we'll put them together in a simple but useful program.

NOTE

Besides sequential data files, two other file types are recorded sequentially on a diskette. They are program files, and user files. When you save a program on a diskette, it is saved in order from beginning to end, just like the information in a sequential data file. The main difference is in the commands you use to access it. User files can be even more similar to sequential data files. User files are almost never used, but like program files, they could be treated as though they were sequential data files and some can be accessed with the same commands.

For the advanced user, the similarity of the various file types offers the possibility of reading a program file into the computer a byte (character) at a time and rewriting it to the diskette in a modified form.

OPENING A FILE

One of the most powerful tools in Commodore BASIC is the OPEN statement. With it, you may send data almost anywhere, much like a telephone switchboard. As you might expect, a command that can do this much is fairly complex. You have already used OPEN statements regularly in some of your diskette commands.

Before you study the format of the OPEN statement, let's review some of the possible devices in a Commodore computer system:

Device#:	Name:	Used for:
0	Keyboard	Receiving input from the computer operator
1	DATASSETTE™	Sending and receiving information from cassette
2	RS232	Sending and receiving information from a modem
3	Screen	Sending output to a video display
4,5	Printer	Sending output to a hard copy printer
8,9,10,11	Disk drive	Sending and receiving information from diskette

Because of the flexibility of the OPEN statement, it is possible for a single program statement to contact any one of these devices, or even others, depending on the value of a single character in the command. If the character is kept in a variable, the device can even change each time that part of the program is used, sending data alternately and with equal ease to diskette, cassette, printer and screen.

NOTE

In the last chapter you learned how to check for disk errors after disk commands in a program. It is equally important to check for disk errors after using file-handling statements. Failure to detect a disk error before using another file-handling statement could cause loss of data, and failure of the BASIC program.

The easiest way to check the disk is to follow all file-handling statements with a GOSUB statement to an error check subroutine.

EXAMPLE:

```
BASIC 7.0
840 DOPEN#4,"DEGREE DAY DATA",D0,U8,W
850 GOSUB 59990: REM CHECK FOR DISK ERRORS

BASIC 2.0
840 OPEN 4,8,4,"0:DEGREE DAY DATA,S,W"
850 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

FORMAT FOR THE DISK OPEN STATEMENT FOR SEQUENTIAL FILES:

BASIC 7.0
DOPEN#file#, "file name" [,Ddrive#] [,Udevice#] [,W]

BASIC 2.0
OPEN file #, device #, channel #,"drive #:file name,file type,-direction"

where:

"file #" is an integer (whole number) between 1 and 255. Do not open a disk file with a file number greater than 127 it will cause severe problems. After the file is open, all other file commands will refer to it by the number given here. Only one file can use any given file number at a time.

"device #" is the number, or primary address, of the device to be used. This number is an integer in the range 8-11, and is normally 8 on the 1581.

"channel #" is a secondary address, giving further instructions to the selected device about how further commands are to be obeyed. In disk files, the channel number selects a particular channel along which

communications for this file can take place. The possible range of disk channel numbers is 0-15, but 0 is reserved for program loads, 1 for program saves, and 15 for the disk command channel. Also be sure that no two disk files have the same channel number unless they will never be open at the same time. (One way to do this is to make the channel number for each file the same as its file number.)

"drive #" is the drive number, always 0 on the 1581. Do not omit it, or you will only be able to use two channels at the same time instead of the normal maximum of three. If any pre-existing file of the same name is to be replaced, precede the drive number with the "at" sign (@) to request OPEN-WITH-REPLACE.

"file name" is the file name, maximum length 16 characters. Pattern matching characters are allowed in the name when accessing existing files, but not when creating new ones.

"file type" is the file type desired: S = sequential, P = program, U = user, A = append and L = length of a relative file.

"direction" is the type of access desired. There are three possibilities: R = read, W = write, and M = modify. When creating a file, use "W" to write the data to diskette. When viewing a completed file, use "R" to read the data from diskette. Only use the "M" (modify) option as a last ditch way of reading back data from an improperly closed (Splat) file. If you try this, check every byte as it is read to be sure the data is still valid, as such files always include some erroneous data, and have no proper end.

"file type" and "direction" don't have to be abbreviated. They can be spelled out in full for clarity in printed listings.

"file #", "device #" and "channel #" must be valid numeric constants, variables or expressions. The rest of the command must be a valid string literal, variable or expression.

"w" is an option that must be specified to write the sequential file, or the file will be opened to read.

The maximum number of files that may be open simultaneously is 10, including all files to all devices. The maximum number of sequential disk files that can be open at once is three (or two if you neglect to include the drive number in your OPEN statement), plus the command channel.

EXAMPLES OF OPENING SEQUENTIAL FILES:

To create a sequential file of phone numbers, you could use:

```
BASIC 7.0: DOPEN#2,"PHONES",D0,U8,W
BASIC 2.0: OPEN 2,8,2,"0:PHONES,SEQUENTIAL,WRITE"
or
OPEN 2,8,2,"0:PHONES,S,W"
```

On the chance you've already got a "PHONES" file on our diskette, you can avoid a "FILE EXISTS" error message by doing an @OPEN

```
BASIC 7.0: DOPEN#2, "@PHONES", D0,U8,W
BASIC 2.0: OPEN 2,8,2,"@0:PHONES,S,W"
```

This erases all your old phone numbers, so make sure that any information that may be deleted is of no importance. After writing our phone file, remove the diskette and turn off the system. To recall the data in the file, reopen it with something like:

```
BASIC 7.0: DOPEN#8,"PHONES",D0,U8
BASIC 2.0: OPEN 8,8,8,"0:PHONES,S,R"
```

It doesn't matter whether the file and channel numbers match the ones we used before, but the file name does have to match. It's possible to use an abbreviation form of the file name, if there are no other files that would have the same abbreviation:

```
BASIC 7.0: DOPEN#10,"PH*",D0,U8
BASIC 2.0: OPEN 10,8,6,"0:PH*,S,R"
```

If you have too many phone numbers, they might not fit in one file. In that case, use several similar file names and let a program choose the correct file.

```
BASIC 7.0:
100 INPUT "WHICH PHONE FILE (1-3)";PH
110 IF PH<>1 AND PH<>2 AND PH<>3 THEN 100
120 DOPEN#4,"PHONE"+STR$(PH),D0,U8

BASIC 2.0:
100 INPUT "WHICH PHONE FILE (1-3)";PH
110 IF PH<>1 AND PH<>2 AND PH<>3 THEN 100
120 OPEN 4,8,2,"PHONE"+STR$(PH)+",S,R"
```

You can omit the drive number on an OPEN command to read a file. Doing so allows those with dual drives to search both diskettes for the file.

FORMAT FOR THE PRINT# COMMAND:

PRINT#file #,data list

where "file #" is the same file number given in the desired file's current OPEN statement. During any given access of a particular file, the file number must remain constant because it serves as a shorthand way of relating all other file-handling commands back to the correct OPEN statement. Given a file number, the computer can look up everything else about a file that matters.

The "data list" is the same as for a PRINT statement - a list of constants, variables and/or expressions, including numbers, strings or both. However, it's better if each PRINT# statement to disk include only one data item. If you wish to include more items, they should be separated by a carriage return character, not a comma. Semicolons are permitted, but not recorded in the file, and do not result in any added spaces in the file. Use them to separate items in the list that might otherwise be confused, such as a string variable immediately following a numeric variable.

NOTE

Do not leave a space between PRINT and #, and do not abbreviate the command as ?#. The correct abbreviation for PRINT# is pR.

EXAMPLES:

To record a few grades for John Paul Jones, using a sequential disk file #1 previously opened for writing, use:

```
200 FOR CLASS = 1 TO COURSES
210 PRINT#1,GRADE$(CLASS)
220 GOSUB 59990:REM CHECK FOR DISK ERRORS
320 NEXT CLASS
```

assuming your program includes an error check subroutine like the one in the last chapter.

In using PRINT#, there is an exception to the requirement to check for disk errors after every file-handling statement. When using PRINT#, a single check after an entire set of data has been written will still detect the error, so long as the check is made before any other file-handling statement or disk command is used. You may be familiar with PRINT statements in which several items follow each other:

```
400 PRINT NAME$,STREET$,CITY$
```

To get those same variables onto sequential disk file number 5 instead of the screen, the best approach would be to use three separate PRINT# statements, as follows:

```
400 PRINT#5,NAME$
410 PRINT#5,STREET$
420 PRINT#5,CITY$
```

If you need to combine them, here is a safe way to do it:

```
400 PRINT#5,NAME$;CHR$(13);STREET$;CHR$(13);CITY$
```

CHR$(13) is the carriage return character, and has the same effect as putting the print items in separate lines. If you do this often, some space and time may be saved by previously defining a variable as equal to CHR$(13):

```
10 CR$=CHR$(13)
400 PRINT#5,NAME$;CR$;STREET$;CR$;CITY$
```

The basic idea is that a proper sequential disk-file write, if redirected to the screen, will display only one data item per line, with each succeeding item on the next line.

CLOSING A FILE

After you finish using a data file, it is extremely important that you CLOSE it. During the process of writing a file, data is accumulated in a memory buffer, and only written out to the diskette when the buffer fills.

Working this way, there is almost always a small amount of data in the buffer that has not been written to diskette yet, and which would simply be lost if the computer system were turned off. Similarly, there are diskette housekeeping matters, such as updating the BAM (Block Availability Map) of sectors used by the current file, which are not performed during the ordinary course of writing a file. This is the reason for having a CLOSE statement. When you are done with a file, the CLOSE statement will write the rest of the data buffer out to diskette, update the BAM, and complete the file's entry in the directory. Always close a data file when you are done using it. Failure to do so may cause loss of the entire file.

However, do not close the disk command channel until all other files have been closed. The command channel should be the first file opened, and the last file closed in any program.

FORMAT FOR THE CLOSE STATEMENT

BASIC 7.0: DCLOSE#file# [,Udevice#]
BASIC 2.0: CLOSE file #

where "file #" is the same file number given in the desired file's current OPEN statement.

EXAMPLES:

To close the data file #5 used as an example on the previous page, use:

BASIC 7.0: DCLOSE#5
BASIC 2.0: CLOSE 5

In BASIC 7.0, when the DCLOSE statement is used alone (no# or file# parameters), it closes all disk files at once. With a bit of planning, the same can be done via a program loop. Since there is no harm in closing a file that wasn't open, close every file you even think might be open before ending a program. If you always gave your files numbers between 1 and 5, you could close them all with

```
9950 FOR I=1 TO 5
9960 CLOSE I
9970 GOSUB 59990:REM CHECK FOR DISK ERRORS
9980 NEXT I
```

assuming your program includes an error check subroutine like the one in the last chapter.

READING FILE DATA: USING INPUT#

Once information has been written properly to a diskette file, it may be read back into the computer with an INPUT# statement. Just as the PRINT# statement is much like the PRINT statement, INPUT# is nearly identical to INPUT, except that the list of items following the command word comes from a particular file instead of the keyboard. Both statements are subject to the same limitations—halting input after a comma or colon, not accepting data items too large to fit in BASIC's input buffer, and not accepting non-numeric data into a numeric variable.

FORMAT FOR THE INPUT# STATEMENT

INPUT#file #,variable list

where "file #" is the same file number given in the desired file's current OPEN statement, and "variable list" is one or more valid BASIC variable names. If more than one data element is to be input by a particular INPUT# statement, each variable name must be separated from others by a comma.

EXAMPLES:

To read back in the grades written with the PRINT# example, use:

```
300 FOR CLASS = 1 TO COURSES
310 INPUT#1,GRADE$(CLASS)
320 GOSUB 59990:REM CHECK FOR DISK ERRORS
330 NEXT CLASS
```

assuming your program includes an error check subroutine like the one in the last chapter.

To read back in the address data written by another PRINT# example, it is safest to use:

```
800 INPUT#5,NAME$
810 GOSUB 59990:REM CHECK FOR DISK ERRORS
820 INPUT#5,STREET$
830 GOSUB 59990:REM CHECK FOR DISK ERRORS
840 INPUT#5,CITY$
850 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

but many programs cheat on safety a bit and use

```
800 INPUT#5,NAME$,STREET$,CITY$
810 GOSUB 59990:REM CHECK FOR DISK ERRORS
```

This is done primarily when top speed in the program is essential, and there is little risk of reading improper data from the file.

MORE ABOUT INPUT#

After you begin using data files regularly, you may encounter two BASIC error messages. They are "STRING TOO LONG ERROR" and "FILE DATA ERROR". Both are likely to halt your program at an

INPUT# statement, but may also have been caused by errors in a PRINT# statement when the file was written.

"String Too Long" Error

A BASIC string may be up to 255 characters long, although the longest string you can enter via a single Input statement is just under two lines of text. This lower limitation is due to the size of the input buffer in Commodore's serial bus computers. The same limit applies to INPUT# statements. If a single data element (string or number) being read from a disk file into an INPUT# statement contains more than 88 (BASIC 2) and 160 (BASIC 7) characters, BASIC will halt with a "STRING TOO LONG ERROR."

"File Data" Error

The other error message "FILE DATA ERROR" is caused by attempting to read a non-numeric character into a numeric variable. To a computer, a number is the characters 0 through 9, the "+" and "−" signs, the decimal point (.), the SPACE character, and the letter "E" used in scientific notation. If any other character appears in an INPUT# to a numeric variable, "FILE DATA ERROR" will be displayed and the program will halt. The usual causes of this error are a mismatch between the order in which variables are written to and read from a file, a missing carriage return within a PRINT# statement that writes more than one data item, or a data item that includes either a comma or a colon without a preceding quotation mark. Once a file data error has occurred, you should correct it by reading the data item into a string variable, and converting it back to a number with the BASIC VAL() statement after removing non-numeric characters with the string functions described in your computer user's manual.

Commas (,) and Colons (:)

As suggested before, commas and colons can cause trouble in a file, because they delimit (end) the data element in which they appear and cause any remaining characters in the data element to be read into the next INPUT# variable. They have the same effect in an INPUT statement, causing the common "EXTRA IGNORED" error message. However, sometimes you really need a comma or colon within a data element, such as a name written as "Last, First." The cure is to precede such data elements with a quotation mark. After a quotation mark, in either an INPUT or INPUT# statement, all other characters except a carriage return or another quotation mark are accepted as part of the current data element.

EXAMPLES:

To force a quotation mark into a data element going to a file, append a CHR$(34) to the start of the data element. For example:

```
PRINT#2,CHR$(34)+"DOE, JOHN"
```

or

```
PRINT#2,CHR$(34);"DOE, JOHN"
```

If you do this often, some space and time may be saved by previously defining a variable as equal to CHR$(34) as we did earlier with CHR$(13):

```
20 QT$=CHR$(34)
...
400 PRINT#5,QT$+NAME$
```

In each case, the added quotation mark will be stripped from the data by the INPUT or INPUT# statement, but the comma or colon will remain part of the data.

NUMERIC DATA STORAGE ON DISKETTE

Up to this point we have discussed string data storage, now let's look at numeric storage.

Inside the computer, the space occupied by a numeric variable depends only on its type. Simple numeric variables use seven bytes (character locations) of memory. Real array variables use five bytes per array element, and integer array elements use two bytes each. In contrast, when a numeric variable or any type is written to a file, the space it occupies depends entirely on its length, not its type. This is because numeric data is written to a file in the form of a string, as if the STR$() function had been performed on it. The first character will be a blank space if the number is positive, and a minus sign (−) if the number is negative. Then comes the number, digit-by-digit. The last character is a cursor right character.

This format allows the disk data to be read back into a string or numeric variable later. It is, however, wasteful of disk space, and it can be difficult to anticipate the space required by numbers of unknown length. For this reason, some programs convert all numeric variables into strings before writing them to diskette, and use string functions to remove any unneeded characters in advance. Doing so still allows

those data elements to be read back into a numeric variable by INPUT# later, although file data errors may be avoided by reading all data in as strings, and converting to numbers using the VAL () function after the information is inside the computer.

For example, "N$=RIGHT$(STR$(N),LEN(STR$(N))−1)" will convert a positive number N into a string N$ without the usual leading space for its numeric sign. Then instead of writing PRINT#5,N, you would use PRINT#5,N$.

READING FILE DATA: USING GET#

The GET# statement retrieves data from the disk drive, one character at a time. Like the similar keyboard GET statement in BASIC, it only accepts a single character into a specified variable. However, unlike the GET statement, it doesn't just fall through to the next statement if there is no data to be gotten. The primary use of GET# is to retrieve from diskette any data that cannot be read into an INPUT# statement, either because it is too long to fit in the input buffer or because it includes troublesome characters.

FORMAT FOR THE GET# STATEMENT:

GET#file#,variable list

where "file #" is the same file number given in the desired file's current OPEN statement, and "variable list" is one or more valid BASIC variable names. If more than one data element is to be input by a particular GET# statement, each variable name must be separated from others by a comma.

In practice, you will almost never see a GET or GET# statement containing more than one variable name. If more than one character is needed, a loop is used rather than additional variables. Also as in the INPUT# statement, it is safer to use string variables when the file to be read might contain a non-numeric character.

Data in a GET# statement comes in byte-by-byte, including such normally invisible characters as the Carriage Return, and the various cursor controls. All but one will be read properly. The exception is CHR$(0), the ASCII Null character. It is different from an empty string (one of the form A$=""), even though empty strings are often referred to as null strings. Unfortunately, in a GET# statement, CHR$(0) is converted into an empty string. The cure is to test for an empty string after a GET#, and replace any that are found with CHR$(0) instead. The first example below illustrates the method.

EXAMPLES:

To read a file that may contain a CHR$(0), such as a machine language program file, you could correct any CHR$(0) bytes with

```
1100 GET#3,G$:IF G$ = "" THEN G$ = CHR$(0)
```

If an overlong string has managed to be recorded in a file, it may be read back safely into the computer with GET#, using a loop such as this

```
3300 B$ = ""
3310 GET#1,A$
3320 IF A$<>CHR$(13) THEN B$ = B$ + A$:GOTO 3310
```

The limit for such a technique is 255 characters. It will ignore CHR$(0), but that may be an advantage in building a text string. If CHR$ (0) is required in the file, then add the following line:

```
3315 If A$ = "" THEN A$ = CHR$(0)
```

GET# may be useful in recovering damaged files, or files with unknown contents. The BASIC reserved variable ST (the file STatus variable) can be used to indicate when all of a properly closed file has been read.

```
500 GET#2,S$
510 SU = ST:REM REMEMBER FILE STATUS
520 PRINT S$;
530 IF SU = 0 THEN 500:REM IF THERE'S MORE TO BE READ
540 IF SU<>64 THEN PRINT "STATUS ERROR: ST = ";SU
```

Copying ST into SU is often an unneccessary precaution, but must be done if any other file-handling statement appears between the one which read from the file and the one that loops back to read again. For example, it would be required if line 520 was changed to

```
520 PRINT#1,S$;
```

Otherwise, the file status checked in line 530 would be that of the write file, not the read file.

The following table applies to single errors or a combination of two or more errors.

POSSIBLE VALUES OF THE FILE STATUS VARIABLE "ST," AND THEIR MEANINGS

IF ST =	THEN
0	All is OK
1	Receiving device was not available (time out on talker)
2	Transmitting device was not available (time out on listener)
4	Cassette data file block was too short
8	Cassette data file block was too long
16	Unrecoverable read error from cassette, verify error
32	Cassette checksum error—one or more faulty characters were read
64	End of file reached (EOI detected)
128	Device not present, or end of tape mark found on cassette

DEMONSTRATION OF SEQUENTIAL FILES
(BASIC 7.0)

Use the following program for your first experiments with sequential files. Comments have been added to help you better understand it.

```
150 CR$=CHR$(13)                    Make a carriage return variable
170 PRINT CHR$(147):REM CLEAR
    SCREEN
190 PRINT "** WRITE A FILE **"
210 PRINT
220 DOPEN #2,"@SEQ FILE",W          Open demo file with replace
230 GOSUB 500                       Check for disk errors
240 PRINT"ENTER A WORD,
    THEN A NUMBER"
250 PRINT"OR 'END,0' TO STOP"
260 PRINT
270 INPUT A$,B                      Accept a string & number
                                    from keyboard
280 PRINT#2,A$;CR$;B                Write them to the disk file
290 GOSUB 500
300 IF A$<>"END" THEN 270           Until finished
310 PRINT
320 DCLOSE #2                       Tidy up
340 PRINT "** READ SAME FILE
    BACK **"
360 PRINT
370 DOPEN #2,"SEQ FILE"             Reopen same file for reading
380 GOSUB 500
390 INPUT#2,A$,B                    Read next string & number
                                    from file
400 RS=ST                           Remember file status
410 GOSUB 500
420 PRINT A$,B                      Display file contents
430 IF RS=0 THEN 390                until done,
440 IF RS<>64 THEN
    PRINT"STATUS=";RS               unless there's an error
450 DCLOSE #2                       Then quit
460 END
480 REM ** ERROR CHECK S/R **
500 IF DS>0 THEN PRINT DS$:STOP
510 RETURN
```

CHAPTER 5
RELATIVE DATA FILES

Sequential files are very useful when you're just working with a continuous stream of data — i.e., information that can be read or written all at once. However, sequential files are not useful in some situations. For example, after writing a large list of mail labels, you wouldn't want to have to reread the entire list each time you need a person's record. Instead, you need some kind of random access, a way to get to a particular label in your file without having to read through all those preceding it.

As an example, compare a record turntable with a cassette recorder. You have to listen to a cassette from beginning to end, but a turntable needle can be picked up at any time, and instantly moved to any spot on the record. Your disk drive works like a turntable in that respect. In this chapter you will learn about a type of file that reflects this flexibility.

Actually, two different types of random access files may be used on Commodore disk drives: relative files and random files. Relative files are much more convenient for most data handling operations, but true random access file commands are also available to advanced users, and will be discussed in the next chapter.

FILES, RECORDS, AND FIELDS

When learning about sequential files, you did not worry about the organization of data within a file, so long as the variables used to write the file matched up properly with those which read it back into the computer. But in order for relative access to work, you need a more structured and predictable environment for our data.

The structure you will use is similar to that used in the traditional filing cabinet. In a traditional office, all customer records might be kept in a single file cabinet. Within this file, each customer has a personal record in a file folder with their name on it, that contains everything the office knows about that person. Likewise, within each file folder, there may be many small slips of paper, each containing one bit of information about that customer, such as a home phone number or the date of the most recent purchase.

In a computerized office, the file cabinet is gone, but the concept of a file containing all the information about a group or topic remains. The file folders are gone too, but the notion of subdividing the file into individual records remains. The slips of paper within the personal

records are gone too, replaced by subdivisions within the records, called fields. Each field is large enough to hold one piece of information about one record in the file. Thus, within each file there are many records, and within each record there are typically many fields.

A relative file takes care of organizing the records for you, numbering them from 1 to the highest record number, by ones, but the fields are up to you to organize. Each record will be of the same size, but the 1581 won't insist that they all be divided the same way. On the other hand, they normally will be subdivided the same way, and if it can be known in advance exactly where each field starts within each record, there are even fast ways to access a desired field within a record without reading through the other fields. As all of this implies, access speed is a primary reason for putting information into a relative disk file. Some well-written relative file programs are able to find and read the record of one desired person out of a thousand in under 15 seconds, a feat no sequential file program could match.

FILE LIMITS

With relative files, you don't have to worry about exactly where on the diskette's surface a given record will be stored, or whether it will fit properly within the current disk sector, or need to be extended onto the next available sector. DOS takes care of all that for you. All you need to do is specify how long each record is, in bytes, and how many records you will need. DOS will do the rest, and organize things in such a way that it can quickly find any record in the file, as soon as it is given the record number (ordinal position within the file).

The only limit that will concern you is that each record must be the same size, and the record length you choose must be between 2 and 254 characters. Naturally the entire file also has to fit on your diskette, along with any other existing file(s).

CREATING A RELATIVE FILE

When a relative file is to be used for the first time, its Open statement will create the file; after that, the Open statement is used to reopen the file for both reading and writing.

FORMAT STATEMENT TO OPEN A RELATIVE FILE:

BASIC 7.0: DOPEN # file #, "file name", L record length
[,Ddrive #] [,Udevice #]
BASIC 2.0: OPEN file #, device #, channel #, "drive #: file name,
L," + CHR$ (record length)

where "file #" is the file number, normally an integer between 1 and 127; "device #" is the device number to be used, normally 8 on the 1581; "channel #" selects a particular channel along which communications for this file can take place, normally between 2 and 14; "drive #" is the drive number, always 0 on the 1581; and "file name" is the name of the file, with a maximum length of 16 characters. Pattern matching characters are allowed in the name when accessing an existing file, but not when creating a new one. The record length is the size of each record within the file in bytes used, including carriage returns, quotation marks and other special characters.

NOTE

- Do not precede the file name (in BASIC 7.0) or the drive number (in BASIC 2.0) with the "at" sign (@); there is no reason to replace a relative file.
- L record length (in BASIC 7.0) or, L ,"+CHR$(record length) (in BASIC 2.0) is only required when a relative file is first created, though it may be used later, so long as the record length is the same as when the file was first created. Since relative files may be read from or written to alternately and with equal ease, there is no need to specify Read or Write mode when opening a relative file.
- "file #", "device #" and "channel #" must be valid numeric constants, variables or expressions. The rest of the command must be a valid string literal, variable or expression. In BASIC 7.0 DOPEN, whenever a variable or expression is used as a file name it must be surrounded by parentheses.

EXAMPLES:

To create or reopen a relative file named "GRADES", of record length 100, use:

```
BASIC 7.0: DOPEN#2,"GRADES",L100,D0,U8
BASIC 2.0: OPEN 2,8,2,"GRADES,L,"+CHR$(100)
```

To reopen an unknown relative file of the user's choice that has already been created, use:

```
BASIC 7.0: 200 INPUT"WHICH FILE";FI$
           210 DOPEN#5,(FI$),D0,U8
```

```
BASIC 2.0: 200 INPUT"WHICH FILE";FI$
           210 OPEN 5,8,5,FI$
```

USING RELATIVE FILES: RECORD# COMMAND

When a relative file is opened for the first time, it is not quite ready for use. Both to save time when using the file later, and to assure that the file will work reliably, it is necessary to create several records before closing the file for the first time. At a minimum, enough records to fill more than two disk sectors (512 bytes) should be written. In practice, most programs go ahead and create as many records as the program is eventually expected to use. That approach has the additional benefit of avoiding such problems as running out of room on the diskette before the entire file is completed.

If you simply begin writing data to a just-opened relative file, it will act much like a sequential file, putting the data elements written by the first PRINT# statement in Record #1, those written by the second PRINT# statement in record #2 and so on. This means each record must be written by a single PRINT# statement, using embedded carriage returns within the data to separate fields that will be read in via one or more INPUT# statements later. However, it is far better to explicitly specify which record number is desired via a RECORD# command to the disk. This allows you to access records in any desired order, hopping anywhere in a file with equal ease.

FORMAT FOR THE RECORD# COMMAND:

```
BASIC 7.0: RECORD # file #, record number [,offset]
BASIC 2.0: PRINT#15, "P"+CHR$ (channel #+96)+CHR$
           (<record #)+CHR$(>record #)+CHR$(offset)
```

where "file #" is the file # specified in the current DOPEN statement for the specified file, "record number" is the desired record number, "channel #" is the channel number specified in the current OPEN statement for the specified file, "<record #" is the low byte of the desired record number, expressed as a two-byte integer, ">record #" is the high byte of the desired record number, and an optional "offset" value, if present, is the byte within the record at which a following Read or Write should begin.

To fully understand this command, you must understand how most integers are stored in computers based on the 6502 and related microprocessors. In the binary arithmetic used by the microprocessor, it is possible to express any unsigned integer from 0-255 in a single

byte. It is also possible to store any unsigned integer from 0-65535 in two bytes, with one byte holding the part of the number that is evenly divisible by 256, and any remainder in the other byte. In machine language, such numbers are written backwards, with the low-order byte (the remainder) first, followed by the high-order byte. In assembly language programs written with the Commodore Assembler, the low part of a two-byte number is indicated by preceding its label with the less-than character (<). Similarly, the high part of the number is indicated by greater-than (>).

NOTE

To avoid the remote possibility of corrupting relative file data, it is necessary to give RECORD# command once before the Read or Write access and once after the access.

Although this is not necessary for the 1581, other Commodore drives require it. To make your programs compatible with those other drives, it's a good idea to use it.

EXAMPLES:

In BASIC 7.0, to position the record pointer for file #2 to record number 3, type:

```
RECORD#2,3
```

In BASIC 2.0, to position the record pointer for channel #2 to record number 3, type:

```
PRINT #15, "P" +CHR$(98) +CHR$(3) + CHR$(0)
```

The CHR$(98) comes from adding the constant (96) to the desired channel number (2). (96+2=98) Although the command appears to work even when 96 is not added to the channel number, the constant is normally added to maintain compatibility with the way RECORD# works in BASIC 7.0.

Since 3 is less than 256, the high byte of its binary representation is 0, and the entire value fits into the low byte. Since you want to read or write from the beginning of the record, no offset value is needed.

Since these calculations quickly become tedious, most programs are written to do them for you. Here is an example of a program which

inputs a record number and converts it into the required low-byte/high-byte form:

```
450 INPUT"RECORD NUMBER DESIRED";RE
460 IF RE<1 OR RE>65535 THEN 450
470 RH=INT(RE/256)
480 RL=RE-256*RH
490 PRINT#15, "P" +CHR$(98) + CHR$(RL) + CHR$(RH)
```

Assuming RH and RL are calculated as in the previous example, programs may also use variables for the channel, record, and offset required:

```
570 INPUT "CHANNEL, RECORD, & OFFSET DESIRED";CH,RE,OF
630 PRINT#15, "P" +CHR$
(CH+96)+CHR$(RL)+CHR$(RH)+CHR$(OF)
```

COMPLETING RELATIVE FILE CREATION

Now that you have learned how to use both the Open and Record# commands, you are almost ready to properly create a relative file. The only additional fact you need to know is that CHR$(255) is a special character in a relative file. It is the character used by the DOS to fill relative records as they are created, before a program fills them with other information. Thus, if you want to write the last record, you expect to need in your file with dummy data that will not interfere with your later work, CHR$(255) is the obvious choice. Here is how it works in an actual program which you may copy for use in your own relative file programs.

BASIC 2.0:

```
1020 OPEN 15,8,15                          Open command channel
1380 INPUT"ENTER RELATIVE FILE NAME";FI$   Select file parameters
1390 INPUT"ENTER MAX. # OF RECORDS";NR
1400 INPUT"ENTER RECORD LENGTH";RL
1410 OPEN 1,8,2,"0:"+FI$+",L,"+CHR$(RL)    Begin to create desired
                                           file
1420 GOSUB 59990                           Check for disk errors
1430 RH=INT(NR/256)                        Calculate length values
1440 RL=NR-256*RH
1450 PRINT#15,"P"+CHR$(96+2)+
     CHR$(RL)+CHR$(RH)                     Position to last record
                                           number
1455 PRINT#15,"P"+CHR$(96+2)+              Re-position for safety
     CHR$(RL)+CHR$(RH)
1460 GOSUB 59990
1470 PRINT#1,CHR$(255);                    Send default character to
                                           it
```

```
1480 GOSUB 59990
1500 GOSUB 59990
1510 CLOSE 1                          Now the file can be safely
                                      closed
1520 GOSUB 59990
9980 CLOSE 15                         And the command chan-
                                      nel closed
9990 END                              Before we end the pro-
                                      gram

59980 REM CHECK DISK SUBROUTINE
59990 INPUT#15,EN,EM$,ET,ES
60000 IF EN>1 AND EN<>50 THEN PRINT   Error check subroutine
      EN,EM$,ET,ES:STOP               Ignore "RECORD NOT
                                      PRESENT"
60010 RETURN
```

BASIC 7.0:

```
1380 INPUT"ENTER RELATIVE FILE NAME";FI$   Select file parameters
1390 INPUT"ENTER MAX. # OF RECORDS";NR
1400 INPUT"ENTER RECORD LENGTH";RL
1410 DOPEN#1,(FI$),L(RL)              Begin to create desired
                                      file
1420 GOSUB 60000                      Check for disk errors
                                      Calculate length values
1450 RECORD#1,(NR)                    Position to last record
                                      number
1455 RECORD#1,(NR)
1460 GOSUB 60000
1470 PRINT#1,CHR$(255);               Send default character to
                                      it
1480 GOSUB 60000
1500 GOSUB 60000
1510 CLOSE 1                          Now the file can be safely
                                      closed
1520 GOSUB 60000
9980 CLOSE 15                         And the command chan-
                                      nel closed
9990 END                              Before we end the pro-
                                      gram

59980 REM CHECK DISK SUBROUTINE
60000 IF DS>1 AND DS<>50 THEN PRINT   Error check subroutine
      DS,DS$:STOP                     Ignore "RECORD NOT
                                      PRESENT"
60010 RETURN
```

Two lines require additional explanation. When line 1470 executes, the disk drive will operate for up to several minutes, creating all the records in the file, up to the maximum record number you selected in line 1390. This is normal, and only needs to be done once. During the process you may hear the drive motor turning and an occasional slight click as the head steps from track to track. Second, line 60000 above is

different from the equivalent line in the error check subroutine given earlier. Here disk error number 50 is specifically ignored, because it will be generated when the error channel is checked in line 1460. Ignore it because not having a requested record would only be an error if that record had been created previously.

EXPANDING A RELATIVE FILE

If you underestimate your needs and want to expand a relative file later, simply request the record number you need, even if it doesn't currently exist in the file. If there is no such record yet, DOS will create it as soon as you try to write information in it, and also automatically create any other missing records below it in number. When the first record beyond the current end record is written, the DOS returns "50, RECORD NOT PRESENT" error. This is expected and correct.

WRITING RELATIVE FILE DATA

The commands used to read and write relative file data are the same PRINT#, INPUT#, and GET# commands used in the preceding chapter on Sequential files. Each command is used as described there. However, some aspects of relative file access do differ from sequential file programming, and we will cover those differences here.

DESIGNING A RELATIVE RECORD

As stated earlier in this chapter, each relative record has a fixed length, including all special characters. Within that fixed length, there are two popular ways to organize various individual fields of information. One is free-format, with individual fields varying in length from record to record, and each field separated from the next by a carriage return character (each of which does take up one character space in the record). The other approach is to use fixed-length fields, that may or may not be separated by carriage returns. If fixed length fields are not all separated by carriage returns, you will either need to be sure a carriage return is included within each 88-character portion of the record (88 is for BASIC 2, 160 is for BASIC 7). If this is not done, you will have to use the GET# command to read the record, at a significant cost in speed.

Since each relative record is most easily written by a single PRINT# statement, the recommended approach is to build a copy of the current record in memory before writing it to disk. It can be collected into a single string variable with the help of BASIC's many

string-handling functions, and then all written out at once from that variable.

Here is an example. If we are writing a 4-line mail label, consisting of 4 fields named "NAME," "STREET," "CITY & STATE," and "ZIP CODE," and have a total record size of 87 characters, we can organize it in either of two ways:

WITH FIXED LENGTH FIELDS

Field Name	Length
NAME	27 characters
STREET	27 characters
CITY & STATE	23 characters
ZIP CODE	10 characters
Total length	87 characters

WITH VARIABLE LENGTH FIELDS

Field Name	Length
NAME	31 characters
STREET	31 characters
CITY & STATE	26 characters
ZIP CODE	11 characters
Potential length	99 characters
Edited length	87 characters

With fixed length records, the field lengths add up to exactly the record length. Since the total length is just within the Input buffer size limitation, no carriage return characters are needed. With variable length records, you can take advantage of the variability of actual address lengths. While one name contains 27 letters, another may have only 15, and the same variability exists in street and city lengths. Although variable length records lose one character per field for carriage returns, they can take advantage of the difference between maximum field length and average field length. A program that uses variable record lengths must calculate the total length of each record as it is entered, to be sure the total of all fields doesn't exceed the space available.

WRITING THE RECORD

Here is an example of program lines to enter variable length fields for the above file design, build them into a single string, and send them to record number RE in file number 3 (assumed to be a relative file that uses channel number 3).

BASIC 7.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 :
120 DOPEN#3,"MYRELFILE",L88
130 CR$=CHR$(13)
140 INPUT"NAME"; NA$
150 IF LEN(A$)>30 THEN 140
160 INPUT"STREET";SA$
170 IF LEN(SA$)>30 THEN 160
```

```
180 INPUT"CITY & STATE"; CS$
190 IF LEN(CS$)>25 THEN 180
200 INPUT"ZIP CODE";ZP$
210 IF LEN(ZP$)>10 THEN 200
220 DA$=NA$+CR$+SA$+CR$+CS$+CR$;ZP$
230 IF LEN(DA$)<88 THEN 260
240 PRINT"RECORD TOO LONG"
250 GOTO 140
260 :
270 :
280 RECORD#3,(RE),1
290 IFDS=50THENPRINT#3,CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 RECORD#3,(RE),1
340 GOSUB1000
350 DCLOSE3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINTDS$:DCLOSE3:END
```

BASIC 2.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 OPEN 15,8,15
120 OPEN3,8,3,"MYRELFILE,L,"+CHR$(88)
130 CR$=CHR$(13)
140 INPUT"NAME"; NA$
150 IF LEN(A$)>30 THEN 140
160 INPUT"STREET";SA$
170 IF LEN(SA$)>30 THEN 160
180 INPUT"CITY & STATE"; CS$
190 IF LEN(CS$)>25 THEN 180
200 INPUT"ZIP CODE";ZP$
210 IF LEN(ZP$)>10 THEN 200
220 DA$=NA$+CR$+SA$+CR$+CS$+CR$;ZP$
230 IF LEN(DA$)<88 THEN 260
240 PRINT"RECORD TOO LONG"
250 GOTO 140
260 RH=INT(RE/256)
270 RL=RE-256*RH
280 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
290 GOSUB1000:IF EN=50THENPRINT#3,CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
340 GOSUB1000
350 CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,ES
1002 IF EN<20 THEN RETURN
1010 PRINTEM$:CLOSE3:CLOSE15:END
```

To use the above program lines for the version with fixed length fields, we would alter a few lines as follows:

BASIC 7.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 :
120 DOPEN#3,"MYRELFILE",L88
130 BL$="(27 shifted space chars)"
140 INPUT"NAME"; NA$
145 LN=LEN(NA$)
150 IF LEN>27 THEN 140
155 NA$=NA$+LEFT$(BL$,27-LN)
160 INPUT"STREET";SA$
165 LN=LEN(SA$)
170 IF LEN>27 THEN 160
175 SA$=SA$+LEFT$(BL$,27-LN)
180 INPUT"CITY & STATE"; CS$
185 LN=LEN(CS$)
190 IF LEN>23 THEN 180
195 CS$=CS$+LEFT$(BL$,23-LN)
200 INPUT"ZIP CODE";ZP$
205 LN=LEN(ZP$)
210 IF LN>10 THEN 200
215 ZP$=ZP$+LEFT$(BL$,10-LN)
220 DA$=NA$+SA$+CS$+ZP$
260 :
270 :
280 RECORD#3,(RE),1
290 IFDS=50THENPRINT#3,CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 RECORD#3,(RE),1
340 GOSUB1000
350 DCLOSE#3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINT"ERROR:"DS$:DCLOSE#):END
```

BASIC 2.0:

```
100 INPUT"ENTER RECORD NUMBER";RE
110 OPEN 15,8,15
120 OPEN#3,8,3,"MYRELFILE,L,"+CHR$(88)
130 BL$="(27 shifted space chars)"
140 INPUT"NAME"; NA$
145 LN=LEN(NA$)
150 IF LEN>27 THEN 140
155 NA$=NA$+LEFT$(BL$,27-LN)
160 INPUT"STREET";SA$
165 LN=LEN(SA$)
```

```
170 IF LEN>27 THEN 160
175 SA$=SA$+LEFT$(BL$,27-LN)
180 INPUT"CITY & STATE"; CS$
185 LN=LEN(CS$)
190 IF LN>23 THEN 180
195 CS$=CS$+LEFT$(BL$,23-LN)
200 INPUT"ZIP CODE";ZP$
205 LN=LEN(ZP$)
210 IF LN>10 THEN 200
215 ZP$=ZP$+LEFT$(BL$,10-LN)
220 DA$=NA$+SA$+CS$+ZP$
260 RH=INT(RE/256)
270 RL=RE-256*RH
280 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
290 GOSUB1000:IF EN=50THENPRINT#3)CHR$(255):GOSUB1000:GOTO280
300 GOSUB1000
310 PRINT#3,DA$
320 GOSUB1000
330 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
340 GOSUB1000
350 GOSUB1000:CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,E
1002 IF EN<20 THEN RETURN
1010 PRINT"ERROR:"EM$:CLOSE3:CLOSE15:END
```

If field contents vary in length, variable field lengths are often preferable. On the other hand, if the field lengths are stable, fixed field lengths are preferable. Fixed length fields are also required if you want to use the optional offset parameter of the RECORD# command to point at a particular byte within a record. However, when any part of a record is written, DOS overwrites any remaining spaces in the record. Thus, if you must use the offset option, never update any field in a record other than the last one unless all succeeding fields will also be updated from memory later.

The above programs are careful to match record lengths exactly to the space available. Programs that don't do so will discover that DOS pads short records out to full size with fill characters, and truncates overlong records to fill only their allotted space. When a record is truncated, DOS will indicate error 51, "RECORD OVERFLOW," but short records will be accepted without a DOS error message.

READING A RELATIVE RECORD

Once a relative record has been written properly to diskette, reading it back into computer memory is fairly simple, but the procedure again varies, depending on whether it uses fixed or variable length fields. Here are the program lines needed to read back the

variable fields created above from record number RE in file and channel 3:

BASIC 7.0:

```
10 :
20 DOPEN#3,"MYRELFILE",L88
30 INPUT"ENTER RECORD NUMBER";RE
40 :
50 :
60 RECORD#3,(RE),1
70 GOSUB1000
80 INPUT#3,NA$,SA$,CS$,ZP$
90 GOSUB1000
100 RECORD#3,(RE),1
110 GOSUB1000
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 DCLOSE#3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINT"ERROR:"DS$:DCLOSE#3:END
```

BASIC 2.0:

```
10 OPEN 15,8,15
20 OPEN3,8,3,"MYRELFILE,L,"+CHR$(88)
30 INPUT"ENTER RECORD NUMBER";RE
40 RH=INT(RE/256)
50 RL=RE-256*RH
60 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
70 GOSUB1000
80 INPUT#3,NA$,SA$,CS$,ZP$
90 GOSUB1000
100 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
110 GOSUB1000
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,ES
1002 IF EN<20 THEN RETURN
1002 PRINT"ERROR:"EM$:CLOSE3:CLOSE15:END
```

READY.

Here are the lines needed to read back the version with fixed length fields:

BASIC 7.0:

```
10 :
20 DOPEN#3,"MYRELFILE",L88
30 INPUT"ENTER RECORD NUMBER";RE
40 :
50 :
60 RECORD#3,(RE),1
70 GOSUB1000
80 INPUT#3,DA$
90 GOSUB1000
100 RECORD#3,(RE),1
110 GOSUB1000
112 NA$=LEFT$(DA$,27)
114 SA$=MID$(DA$,28,27)
116 CS$=MID$(DA$,55,23)
118 ZP$=RIGHT$(DA$,10)
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 DCLOSE#3:END
1000 IFDS<20 THEN RETURN
1002 :
1010 PRINT"ERROR:"DS$:DCLOSE#3:END
```

BASIC 2.0:

```
10 OPEN 15,8,15
20 OPEN3,8,3,"MYRELFILE,L"+CHR$(88)
30 INPUT"ENTER RECORD NUMBER";RE
40 RH=INT(RE/256)
50 RL=RE-256*RH
60 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
70 GOSUB1000
80 INPUT#3,DA$
90 GOSUB1000
100 PRINT#15,"P"+CHR$(96+3)+CHR$(RL)+CHR$(RH)+CHR$(1)
110 GOSUB1000
112 NA$=LEFT$(DA$,27)
114 SA$=MID$(DA$,28,27)
116 CS$=MID$(DA$,55,23)
118 ZP$=RIGHT$(DA$,10)
120 PRINTNA$:PRINTSA$
130 PRINTCS$:PRINTZP$
140 CLOSE3:CLOSE15:END
1000 INPUT#15,EN,EM$,ET,ES
1002 IF EN<20 THEN RETURN
1002 PRINT"ERROR:"EM$:CLOSE3:CLOSE15:END
```

READY.

THE VALUE OF INDEX FILES

In the last two chapters you have learned how to use sequential and relative files separately. But they are often used together, with the sequential file used to keep brief records of which name in the relative file is stored in each record number. That way the contents of the sequential file can be read into a string array and sorted alphabetically. After sorting, a technique known as a binary search can be used to quickly find an entered name in the array, and read in or write the associated record in the relative file. Advanced programs can maintain two or more such index files, sorted in differing ways simultaneously.

CHAPTER 6
DIRECT ACCESS COMMANDS

Direct access commands specify individual sectors on the diskette, reading and writing information entirely under your direction. This gives them almost complete flexibility in data-handling programs, but imposes tremendous responsibilities on the programmer. As a result, they are normally used only in complex commercial programs able to properly organize data without help from the disk drive itself.

A far more common use of direct access commands is in utility programs used to view and alter parts of the diskette that are not normally seen directly. For instance, such commands can be used to change the name of a diskette without erasing all of its programs, to lock a program so it can't be erased, or hide your name in a location where it won't be expected.

OPENING A DATA CHANNEL FOR DIRECT ACCESS

When working with direct access data, you need two channels open to the disk: the command channel we've used throughout the book, and another for data. The command channel is opened with the usual OPEN 15,8,15 or equivalent. A direct access data channel is opened much like other files, except that the pound sign (#), optionally followed by a memory buffer number, is used as a file name.

FORMAT FOR DIRECT ACCESS FILE OPEN STATEMENTS:

OPEN file #,device #, channel #, "#buffer #"

where "file #" is the file number, "device #" is the disk's device number, normally 8; "channel #" is the channel number, a number between 2 and 14 not used by other files open at the same time; and "buffer #," if present, is a 0, 1, 2, 3, 4, 5, or 6, specifying the memory buffer within the 1581 to use for this file's data.

EXAMPLES:

To specify which disk buffer to use:

OPEN 4,8,4,"#2"

If you don't specify which to use (OPEN 5,8,5,"#"), the 1581 selects one.

BLOCK-READ

The purpose of a BLOCK-READ is to load the contents of a specified sector into a file buffer. Although the BLOCK-READ command (B-R) is still part of the DOS command set, it is nearly always replaced by the U1 command (See Chapter 6).

FORMAT FOR THE BLOCK-READ COMMAND:

PRINT#15, "U1"; channel #; drive #; track #; sector #

where "channel #" is the channel number specified when the file into which the block will be read was opened, "drive #" is the drive number, and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be read into the file buffer.

ALTERNATE FORMATS:

PRINT#15,"U1:"channel #;drive #;track #;sector #
PRINT#15,"UA:"channel #;drive #;track #;sector #
PRINT#15,"U1:channel #,drive #,track #,sector #"

EXAMPLE:

Here is a complete program to read a sector into disk memory using U1, and from there into computer memory via GET#. (If a carriage return will appear at least once in every 88 characters of data, INPUT# may be used in place of GET#).

```
110 MB=7936:REM $1F00          Define a memory buffer.
120 INPUT"TRACK TO READ";T     Select a track
130 INPUT"SECTOR TO READ";S    and sector.
140 OPEN 15,8,15               Open command channel.
150 OPEN 5,8,5,"#"             Open direct access channel.
160 PRINT#15,"U1";5;0;T;S      Read sector into disk buffer.
170 FOR I=MB TO MB+255         Use a loop to
180 GET#5,A$:IF A$="" "        copy disk buffer.
    THEN A$=CHR$(0)            into computer memory.
190 POKE I,ASC(A$)             Tidy up after.
200 NEXT
210 CLOSE 5:CLOSE 15
220 END
```

As the loop progresses, the contents of the specified track and sector are copied into computer memory, beginning at the address set by variable MB in line 160, and may be examined and altered there.

The DOS always checks that the track and sector parameters of the BLOCK-READ command are within the proper range. If they're not, a "66 ILLEGAL TRACK AND SECTOR" error occurs. In certain instances it might be necessary to access a track and sector that are not within what the DOS considers the proper bounds. This is a special case and, unless absolutely necessary, should be avoided. Nonetheless, there is a command identical in function to "U1" that doesn't check to see if the track and sector parameters are within bounds before attempting to read it. Its format is:

```
PRINT#15,"B-▁";channel #;track #;sector #
(The character following the B- is a shifted R.)
                    or
PRINT#15,"B-";CHR$(210);channel #;track #;sector #
```

BLOCK-WRITE

The purpose of a BLOCK-WRITE is to save the contents of a file buffer into a specified sector. It is thus the reverse of the BLOCK-READ command. Although the BLOCK-WRITE command (B-W) is still part of the DOS command set, it is nearly always replaced by the U2 command.

FORMAT FOR THE BLOCK-WRITE COMMAND:

```
PRINT#15,"U2";channel #;drive #;track #;sector #
```

where "channel #" is the channel number specified when the file into which the block will be read was opened; "drive #" is the drive number; and "track #" and "sector #" are respectively the track and sector numbers that should receive the block of data being saved from the file buffer.

ALTERNATE FORMATS:

```
PRINT#15,"U2:"channel #;drive #;track #;sector #
PRINT#15,"UB:"channel #;drive #;track #;sector #
PRINT#15,"U2:channel #,drive #,track #,sector #"
```

EXAMPLES:

To restore track 40, sector 3 of the directory from the disk buffer filled by a BLOCK-READ, use:

```
PRINT#15,"U2";5;0;40;3
```

You'll return to this example on the next page, after you learn to alter the directory in a useful way.

You can also use a BLOCK-WRITE to write a name in Track 1, Sector 1, a rarely-used sector. This can be used as a way of marking a diskette as belonging to you. Here is a program to do it, using the alternate form of the BLOCK-WRITE command:

```
110 INPUT"YOUR NAME";NA$        Enter a name.
120 OPEN 15,8,15                Open command channel.
130 OPEN 4,8,4,"#"              Open direct access channel.
140 PRINT#4,NA$                 Write name to buffer.
150 PRINT#15,"U2";4;0;1;1       Write buffer to Track 1,
160 CLOSE 4                     Sector 1 of diskette.
170 CLOSE 15                    Tidy up after.
180 END
```

As with the BLOCK-READ command, there is a BLOCK-WRITE command identical in function to "U2" that does not check the track and sector parameters for valid bounds before attempting to write the sector. Its format is:

```
PRINT#15,"B-o";channel #;drive #;track #;sector #
```

(The character after the B- is a shifted W.)

or

```
PRINT#15,"B-";CHR$(215);channel #;track #;sector #
```

THE ORIGINAL BLOCK-READ AND BLOCK-WRITE COMMANDS

Although the BLOCK-READ and BLOCK-WRITE commands are nearly always replaced by the U1 and U2 commands respectively, the original commands can still be used, as long as you fully understand their effects. Unlike U1 and U2, B-R and B-W allow you to read or write less than a full sector. In the case of B-R, the first byte of the selected sector is used to set the buffer pointer (see next section), and deter-

mines how many bytes of that sector are read into a disk memory buffer. A program may check to be sure it doesn't attempt to read past the end of data actually loaded into the buffer, by watching for the value of the file status variable ST to change from 0 to 64. When the buffer is written back to diskette by B-W, the first byte written is the current value of the buffer pointer. Only that many bytes are written into the specified sector. B-R and B-W may thus be useful in working with custom-designed file structures.

FORMAT FOR THE ORIGINAL BLOCK-READ AND BLOCK-WRITE COMMANDS:

PRINT#15,"BLOCK-READ";channel #;drive #;track #;sector #

abbreviated as: PRINT#15,"B-R";channel #;drive #;track #;sector #

and

PRINT#15,"BLOCK-WRITE";channel #;drive #;track #;sector #

abbreviated as: PRINT#15,"B-W";channel #;drive #;track #;sector #

where "channel #" is the channel number specified when the file into which the block will be read was opened, "drive #" is the drive number, and "track #" and "sector #" are the track and sector numbers containing the desired block of data to be partially read into or written from the file buffer.

NOTE

In a true BLOCK-READ, the first byte of the selected sector is used to determine how many bytes of that sector to read into the disk memory buffer. It thus cannot be used to read an entire sector into the buffer, as the first data byte is always interpreted as being the number of characters to read, rather than part of the data.

Similarly, in a true BLOCK-WRITE, when the buffer is written back to diskette, the first byte written is the current value of the buffer pointer. Only that many bytes are written into the specified sector. It cannot be used to rewrite an entire sector onto diskette unchanged, because the first data byte is overwritten by the buffer pointer.

THE BUFFER POINTER

The buffer pointer points to where the next READ or WRITE will begin within a disk memory buffer. By moving the buffer pointer, you can access individual bytes within a block in any order. This allows you to edit any portion of a sector, or organize it into fields, like a relative record.

FORMAT FOR THE BUFFER-POINTER COMMAND:

PRINT#15,"BUFFER-POINTER";channel #;byte

usually abbreviated as: PRINT#15,"B-P";channel #;byte

where "channel #" is the channel number specified when the file reserving the buffer was opened, and "byte" is the character number within the buffer at which to point (from 0 through 255).

ALTERNATE FORMATS:

PRINT#15,"B-P:"channel #;byte
PRINT#15,"B-P:channel #;byte"

EXAMPLE:

Here is a program that locks the first program or file on a diskette. It works by reading the start of the directory (Track 40, Sector 3) into disk memory, setting the buffer pointer to the first file type byte (see Appendix C for details of directory organization), locking it by setting bit 6 and rewriting it.

```
110 OPEN 15,8,15                  Open command channel.
120 OPEN 5,8,5,"#"                Open direct access channel.
130 PRINT#15,"U1";5;0;40;3        Read Track 40, Sector 3.
140 PRINT#15,"B-P";5;2            Point to Byte 2 of the buffer.
150 GET#5,A$:IF A$="" "
  THEN A$=CHR$(0)                 Read it into memory.
160 A=ASC(A$) OR 64               Turn on bit 6 to lock.
170 PRINT#15,"B-P";5;2            Point to Byte 2 again.
180 PRINT#5,CHR$(A);              Overwrite it in buffer.
190 PRINT#15,"U2";5;0;40;3        Rewrite buffer to diskette.
200 CLOSE 5                       Tidy up after.
210 CLOSE 15
220 END
```

After the above program is run, the first file on that diskette can no longer be erased. If you later need to erase that file, rerun the same program, but substitute the revised line 160 below to unlock the file again:

```
160 A=ASC(A$) AND 191          Turn off bit 6 to unlock
```

ALLOCATING BLOCKS

Once you have written something in a particular sector on a diskette with the help of direct access commands, you may wish to mark that sector as "already used", to keep other files from being written there. Blocks thus allocated will be safe until the diskette is validated.

FORMAT FOR BLOCK-ALLOCATE COMMAND:

PRINT#15,"BLOCK-ALLOCATE";drive #; track #;sector #

usually abbreviated as: PRINT#15,"B-A";drive #; track #;sector #

where "drive #" is the drive number, and "track #" and "sector #" are the track and sector containing the block of data to be read into the file buffer.

ALTERNATE FORMAT:

PRINT#15,"B-A:";drive #; track #;sector #

EXAMPLE:

If you try to allocate a block that isn't available, the DOS will set the error message to number 65, NO BLOCK, and set the track and block numbers in the error message to the next available track and block number. Therefore, before selecting a block to write, try to allocate that block. If the block isn't available, read the next available block from the error channel and allocate it instead. However, do not allocate data blocks in the directory track. If the track number returned is 0, the diskette is full.

Here is a program that allocates a place to store a message on a diskette.

```
100 OPEN15,8,15          Open command channel.
110 OPEN5,8,5,"#"        " direct access "
```

```
120 PRINT#5,"I THINK
    THEREFORE I AM"          Write a message to buffer.
130 T=1:S=1                  Start at first track & sector.
140 PRINT#15,"B-A";0;T;S     Try allocating it.
150 INPUT#15,EN,EM$,ET,ES    See if it worked.
160 IF EN=0 THEN 210         If so, we're almost done.
170 IF EN<>65 THEN PRINT     "NO BLOCK"
    EN,EM$,ET,ES:STOP        means already allocated.
180 IF ET=0 THEN PRINT
    "DISK FULL":STOP         If next track is 0, we're out of room.
190 IF ET=40 THEN ET=41:ES=0 Don't allocate the directory.
200 T=ET:S=ES:GOTO 140       Try suggested track & sector next.
210 PRINT#15,"U2";5;0;T;S    Write buffer to allocated sector.
220 PRINT "STORED AT:",T,S   Say where message went
230 CLOSE 5:CLOSE 15         and tidy up.
240 END
```

FREEING BLOCKS

The BLOCK-FREE command is the opposite of BLOCK-ALLOCATE. It frees a block that you don't need any more, for re-use by the DOS. BLOCK-FREE updates the BAM to show a particular sector is not in use, rather than actually erasing any data.

FORMAT FOR BLOCK-FREE COMMAND:

PRINT#15,"BLOCK-FREE";drive #;track #;sector #

abbreviated as: PRINT#15,"B-F";drive #;track #;sector #

where "drive #" is the drive number, and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be read into the file buffer.

ALTERNATE FORMAT:

PRINT#15,"B-F:";drive #;track #;sector #

EXAMPLE:

To free the sector in which we wrote our name in the BLOCK WRITE example, and allocated in the first BLOCK-ALLOCATE example, we could use the following command:

PRINT#15,"B-F";0;1;1

PARTITIONS and SUB-DIRECTORIES

The 1581 allows the user to create partition areas on the disk. Partitions were originally implemented to provide a mechanism for easily protecting a particular section of the disk. That is useful for permanently allocating part of the disk for things such as BOOT sectors, CP/M work area, or reserving space for user defined random files.

Normally, sectors on the disk can be marked as used by setting the appropriate bit in the BAM (most easily done with the BLOCK-ALLOCATE command). That prevents them from being overwritten. A VALIDATE command, however, will de-allocate this area. To protect these special blocks from being de-allocated during a VALIDATE, place them in a user defined partition area. The VALIDATE command in the 1581 automatically skips over file entries that are partition files (file type = CBM), which guarantees the intended area is, and remains, allocated.

Partition areas are given names by the user when first created. They appear in the main directory as file type CBM.

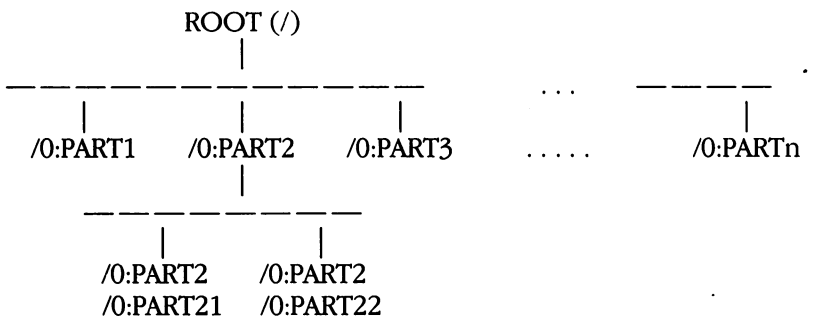
A partition area is created by the following command (file# should be opened to the command channel):

```
PRINT#file#,"/0:partition name,"+CHR$(starting track)+CHR-
$(starting sector)+CHR$(< # of sectors)+CHR$(> # of sec-
tors)+",C"
```

Large enough partitions can also be used as sub-directories. There are, however, certain limitations if a partition area is to be used as a sub-directory area:

1) The partition area must be at least 120 sectors in size.
2) The starting sector must be 0.
3) The ending sector must be a multiple of 40.
4) The area to be allocated cannot contain track 40 (the original system track).

Partitions can also be created with a partition. This means that sub-sub-directories can be created if their partitions meet the above rules. Graphically, it looks like this:



Partition areas which meet the qualifications of being a subdirectory can then be selected by the following command.

PRINT#file#,"/0:partition name"

Once selected, the partition area cannot be used as a sub-directory until it is formatted. The HEADER or NEW commands are used to format this sub-disk area. Make sure that you have successfully selected this partition area before formatting. If not, the wrong directory area will be reformatted. You can check if the area was successfully selected by checking the error channel. If everything went OK, the error channel would read:

02, SELECTED PARTITION,first track #,last track #

If the area you attempt to select does not meet the qualifications of a sub-directory, then the error channel would return:

77, SELECTED PARTITION ILLEGAL,00,00

Only one level of subdirectory can be selected at a time. To get from the ROOT to PART21 you would have to execute the command twice.

PRINT#file#,"/0:PART2"
PRINT#file#,"/0:PART21"

Directories can only be traversed in the forward direction. To get to a sub-directory which is on a node above the presently selected node of the tree, you must select the ROOT directory and work your way down the tree, selecting a branch at a time. To get to the ROOT directory directly from any node type:

PRINT#file#,"/"

When the user selects a particular sub-directory area, it then becomes the default working area. Accesses to the disk for directories, loading files, saving files, etc., will all be done within this area. Files outside of the selected area are effectively invisible.

File and local BAM information for sub-directories are stored within the sub-directory areas themselves. The information is stored on the first allocated track of the partition area, and has the same format as track 40. When creating partitions and sub-directories within sub-directories it is the responsibility of the user to make sure that he doesn't overwrite this information! The DOS only checks to make sure that you don't attempt to overwrite this information for the ROOT directory (track 40). It is up to the user to make sure that this information isn't corrupted in the sub-directories.

Partitioned areas can be freed up simply by SCRATCHING the partition file entry in the appropriate directory. If the partition was being used as a sub-directory, all of the files in that sub-directory will be lost.

USING RANDOM FILES

By combining the commands in this chapter, it is possible to develop a file-handling program that uses random files. What you need to know now is how to keep track of which blocks on the disk such a file has used. (Even though you know a sector has not been allocated by your random file, you must also be sure it wasn't allocated by another unrelated file on the diskette.)

The most common way of recording which sectors have been used by a random file is in a sequential file. The sequential file stores a list of record numbers, with the track, sector, and byte location of each record. This means three channels are needed by a random file: one for the command channel, one for the random data, and the last for the sequential data.

CHAPTER 7
INTERNAL DISK COMMANDS

Expert programmers can give commands that directly alter the workings of the 1581, much as skilled programmers can alter the workings of BASIC inside the computer with Peeks, Pokes and Sys calls. It is also possible to write machine language programs that load and run entirely within the 1581, either by writing them into disk memory from the computer, or by loading them directly from diskette into the desired disk memory buffer. This is similar to loading and running machine language programs in your computer.

As when learning to use Peek, Poke and Sys in your computer, extreme caution is advised in using the commands in this chapter. They are essentially machine language commands, and lack all of BASIC'S safeguards. If anything goes wrong, you may have to turn the disk drive off and on again (after removing the diskette) to regain control. Do not practice these commands on any important diskette. Rather, make a spare copy and work with that. Knowing how to program a 6502 in machine language will help greatly, and you will also need a good memory map of the 1581. A brief 1581 map appears below.

1581 MEMORY MAP

Location	Purpose
0000-00FF	Zero page work area, job queue, variables
0100-01FF	Stack, variables, vectors
0200-02FF	Command buffer, tables, variables
0300-09FF	Data buffers (0-6)
0A00-0AFF	BAM for tracks 0-39
0B00-0BFF	BAM for tracks 40-79
0C00-1FFF	Track cache buffer
4000-5FFF	8520A CIA
6000-7FFF	WD177X FDC
8000-FEFF	32K byte ROM, DOS and controller routines
FF00-FFFF	Jump table, vectors

NOTE

The 1581, as well as other Commodore peripherals, is designed to support interfacing via software command structures. The software commands provided in the 1581 allow for a smooth and controllable interface between the peripheral and CPU. Although Commodore has provided the mechanism enabling users to load and execute their own machine language programs within the 1581 system, please keep in mind that Commodore reserves the right to change ROM, RAM, I/O and hardware structure at any time. Consequently, if the defined software interface is bypassed, future compatibility of the user's machine language software within the 1581 may be in question. The 1581 was not designed primarily as a user programmable device, but Commodore recognizes that certain operations (such as copy protection) cannot be easily achieved without this ability.

If you find it necessary to use machine language within the 1581, use the jump table listed in this chapter and Chapter 10. That will lessen the possibility of incompatibility if a future version of the 1581 changes internally. Also, let the controller work for you on the physical level by requesting its help via the JOB QUEUE. That too will greatly increase the likelihood of future compatibility.

MEMORY-READ

The 6502 has an address space from $0000 — $FFFF. You can get direct access to any location within this by using memory commands. MEMORY-READ allows you to select which byte or bytes to read from disk memory into the computer. The MEMORY-READ command is the equivalent of the BASIC Peek() function, but reads the disk's memory instead of the computer's memory.

NOTE

Unlike other disk commands, those in this chapter cannot be spelled out in full. Thus, M-R is correct, but MEMORY-READ is not a permitted alternate wording.

FORMAT FOR THE MEMORY-READ COMMAND:

PRINT#15,"M-R"CHR$(<address)CHR$(>address)CHR$(# of bytes)

where "<address" is the low order part, and ">address" is the high order part of the address in disk memory to be read. If the optional "# of bytes" is specified, it selects how many memory locations will be read in, from 1-256 (# of bytes = 0 for 256). Otherwise, 1 character will be read.

The next byte read using the GET# statement through channel #15 (the error channel), will be from that address in the disk controller's memory, and successive bytes will be from successive memory locations.

Any INPUT# from the error channel will give peculiar results when you're using this command. This can be cleared up by sending any other command to the disk, except another memory command.

EXAMPLES:

To see how many tries the disk will make to read a particular sector, and whether "bumps" to track one and back will be attempted before declaring the sector unreadable, you can use the following lines. They will read a special variable in the zero page of disk memory, called REVCNT. It is located at $30 hexadecimal.

```
110 OPEN 15,8,15                          Open command channel.
120 PRINT#15,"M-R"CHR$(48)CHR$(0)         Same as G=PEEK(106).
130 GET#15,G$:IF G$="" THEN G$=CHR$(0)
140 G=ASC(G$)
150 B=G AND 128:B$="ON":IF B THEN B$="OFF"   Check bit 7.
170 T=G AND 31:PRINT "# OF TRIES IS";T       Check bits 0-5
180 PRINT "BUMPS ARE";B$                     and give results.
200 CLOSE 15                              Tidy up after.
210 END
```

Here's a more general purpose program that reads one or more locations anywhere in disk memory:

```
110 OPEN15,8,15                          Open command channel.
120 INPUT"# OF BYTES TO READ (0=END)";NL  Enter number of bytes wanted
130 IF NL<1 THEN CLOSE 15:END            unless done.
140 IF NL>255 THEN 120                   or way out of line.
150 INPUT"STARTING AT ADDRESS";AD        Enter starting address.
160 AH=INT(AD/256):AL=AD-AH*256          Convert it into disk form.
170 PRINT#15,"M-R"CHR$(AL)CHR$(AH)       Actual Memory-Read.
    CHR$(NL)
180 FOR I=1 TO NL                        Loop until have all the data,
190 GET#15,A$:IF A$="" THEN A$=CHR$(0)
200 PRINT ASC(A$);                       printing it as we go,
210 NEXT I
220 PRINT
230 GOTO 120                             forever.
```

MEMORY-WRITE

The MEMORY-WRITE command is the equivalent of the BASIC Poke command, but has its effect in disk memory instead of within the computer. M-W allows you to write up to 35 bytes at a time into disk memory. The MEMORY-EXECUTE and some User commands can be used to run any programs written this way.

FORMAT FOR THE MEMORY-WRITE COMMAND:

PRINT#15,"M-W"CHR$(<address)CHR$(>address)CHR$
(# of bytes)CHR$(data byte(s))

where "<address" is the low order part, and ">address" is the high order part of the address in disk memory to begin writing, "# of bytes" is the number of memory locations that will be written (from 1-35), and "data byte" is 1 or more byte values to be written into disk memory, each as a CHR$() value.

EXAMPLES:

We can use this line to turn off the "bumps" when loading DOS-protected programs (i.e., programs that have been protected against being copied by creating and checking for specific disk errors).

PRINT#15,"M-W"CHR$(48)CHR$(0)CHR$(1)CHR$(133)

The following line can be used to recover bad sectors, such as when an important file has been damaged and cannot be read normally.

```
PRINT#15,"M-W"CHR$(48)CHR$(0)CHR$(1)CHR$(31)
```

These two examples may be very useful under some circumstances. They are the equivalent of POKE 48,133 and POKE 48,31 respectively, but in disk memory, not inside the computer. As mentioned in the previous section's first example, location 48 in the 1581 disk drive signifies two separate activities to the drive, all related to error recovery. Bit 7 (the high bit), if set means no "bumps" (don't move the read head to track 1). The bottom six bits are the count of how many times the disk will try to read each sector before and after trying seeks and bumps before giving up. Since 31 is the largest number that can be expressed in six bits, that is the maximum number of tries allowed.

From this example, you can see the value of knowing something about Peeks, Pokes, and machine-language before using direct-access disk commands, as well as their potential power.

MEMORY-EXECUTE

Any routine in disk memory, either in RAM or ROM, can be executed with the MEMORY-EXECUTE command. It is the equivalent of the BASIC Sys call to a machine language program or subroutine, but works in disk memory instead of within the computer.

FORMAT FOR THE MEMORY-EXECUTE COMMAND:

```
PRINT#15,"M-E"CHR$(<address)CHR$(>address)
```

where "<address" is the low order part, and ">address" is the high order part of the address in disk memory at which execution is to begin.

Most uses require intimate knowledge of the inner workings of the DOS, and preliminary setup with other commands, such as MEMORY-WRITE.

The routine should end with an RTS to return control to the 1581.

BLOCK-EXECUTE

This rarely-used command will load a sector containing a machine language routine into a memory buffer from diskette, and execute it

from the first location within the buffer, until a RETURN from Subroutine (RTS) instruction ends the command.

FORMAT FOR THE BLOCK-EXECUTE COMMAND:

PRINT#15,"B-E:";channel #;drive #;track #;sector #

where "channel #" is the channel number specified when the file into which the block will be loaded was opened, "drive #" is the drive number, and "track #" and "sector #" are respectively the track and sector numbers containing the desired block of data to be loaded into the file buffer and executed there.

ALTERNATE FORMATS:

PRINT#15,"B-E:";channel #;drive #;track #;sector #
PRINT#15,"B-E:channel #,drive #,track #,sector #"

EXAMPLES:

Assuming you've written a machine language program onto Track 1, Sector 8 of a diskette, and would like to run it in buffer number 1 in disk memory (starting at $0400 hexadecimal, you could do so as follows:

```
110 OPEN 15,8,15                Open command channel.
120 OPEN 2,8,2,"#1"             Open direct access channel to buffer 1.
130 PRINT#15,"B-E:";2;0;1;8     Load Track 1, Sector 8 in it & execute.
140 CLOSE 2                     Tidy up after.
150 CLOSE 15
160 END
```

USER COMMANDS

Most User commands are intended to be used as machine language JMP or BASIC SYS commands to machine language programs that reside inside the disk memory. However, some of them have other uses as well. The User1 and User2 commands are used to replace the BLOCK-READ and BLOCK-WRITE commands, UI re-starts the 1581 without changing many variables, UJ cold-starts the 1581 almost as if it had been turned off and on again.

User Command	Function
u0	restores default user jump table
u0 + (cmd)	burst utility command (see Chapter 9 Burst Commands)
u1 or ua	block read replacement
u2 or ub	block write replacement
u3 or uc	jump to $0500
u4 or ud	jump to $0503
u5 or ue	jump to $0506
u6 or uf	jump to $0509
u7 or ug	jump to $050c
u8 or uh	jump to $050f
u9 or ui	jump to ($fffa) reset tables
u: or uj	power up vector

By loading these memory locations with another machine language JMP command, such as JMP $0520, you can create longer routines that operate in the disk's memory along with an easy-to-use jump table.

FORMAT FOR USER COMMANDS:

PRINT#15,"Ucharacter";

where "character" defines one of the preset user commands listed above.

EXAMPLES:

PRINT#15,"U:";	Form of DOS RESET command
PRINT#15,"U3";	Execute program at start of buffer 2

UTILITY LOADER

This command loads a user-type (USR) file into the drive RAM. The first two bytes of the file must contain the low and high addresses respectively. The third byte is the amount of characters to follow. In addition, a trailing checksum byte must be included. The load address is the starting address.

FORMAT FOR THE UTILITY LOADER COMMAND

PRINT#15,"&0:filename"

To return from this routine, the program should end with an RTS.

AUTO BOOT LOADER

During some operations (power-up reset, burst INQURE, burst QUERY, an initialize command) the 1581 will automatically look for a file or the disk named 'COPYRIGHT CBM 86' that is a USR type-file. The format of the file is the same as that described previously for the utility loader. If it is present, the file is automatically loaded and executed.

The automatic loading of this file can be disabled by either renaming it, setting the appropriate flag in the BAM sectors (see Appendix C), or by setting a flag variable in RAM to disable further autoboots (see JDEJAVU jump table vector in Chapter 10).

At the end of the autobooted program it should return control to the 1581 via the JCBMBOOTRTN jump table vector.

CHAPTER 8
MACHINE LANGUAGE PROGRAMS

Here is a list of host computer disk-related Kernal ROM subroutines and a practical example of their use in a program that reads a sequential file into memory from disk. Most require advance setup of one or more processor registers or memory locations and all are called with the assembly language JSR command.

For a more complete description as to what each routine does and how parameters are set for each routine, see the Programmer's Reference Guide for your specific computer.

DISK-RELATED KERNAL SUBROUTINES

Label		Address	Function
SETLFS	=	$FFBA	;SET LOGICAL, FIRST & SECOND ADDRESSES
SETNAM	=	$FFBD	;SET LENGTH & ADDRESS OF FILENAME
OPEN	=	$FFC0	;OPEN LOGICAL FILE
CLOSE	=	$FFC3	;CLOSE LOGICAL FILE
CHKIN	=	$FFC6	;SELECT CHANNEL FOR INPUT
CHKOUT	=	$FFC9	;SELECT CHANNEL FOR OUTPUT
CLRCHN	=	$FFCC	;CLEAR ALL CHANNELS & RESTORE DEFAULT I/O
CHRIN	=	$FFCF	;GET BYTE FROM CURRENT INPUT DEVICE
CHROUT	=	$FFD2	;OUTPUT BYTE TO CURRENT OUTPUT DEVICE

```
START   LDA   #4          ;SET LENGTH & ADDRESS
        LDX   #<FNADR     ;OF FILE NAME, LOW
        LDY   #>FNADR     ;& HIGH BYTES
        JSR   SETNAM      ;FOR NAME SETTER
        LDA   #3          ;SET FILE NUMBER
        LDX   #8          ;DISK DEVICE NUMBER
        LDY   #0          ;AND SECONDARY ADDRESS
        JSR   SETLFS      ;AND SET THEM
        JSR   OPEN        ;OPEN 3,8,0,"TEST"
        LDX   #3
        JSR   CHKIN       ;SELECT FILE 3 FOR INPUT
NEXT    JSR   CHRIN       ;GET NEXT BYTE FROM FILE
        BEQ   END         ;UNTIL FINISH OR FAIL
        JSR   CHROUT      ;OUTPUT BYTE TO SCREEN
        JMP   NEXT        ;AND LOOP BACK FOR MORE
;
END     LDA   #3          ;WHEN DONE
        JSR   CLOSE       ;CLOSE FILE
        JSR   CLRCHN      ;RESTORE DEFAULT I/O
        RTS               ;BACK TO BASIC
;
FNADR   .BYT  "TEST"      ;STORE FILE NAME HERE
```

CHAPTER 9
BURST COMMANDS

The Burst Command Instruction Set (BCIS) is a series of powerful, versatile, and complex commands that enables the user to format, read, and write in numerous formats. Burst commands are sent via kernal calls, but the handshaking of data is done by the user for maximum performance. There is no parameter checking, so exercise care when using the BCIS. For instance, if a burst read with an illegal track address is sent to a 1581, the drive will keep trying to find the invalid track. Reading and writing in other formats is automatic if the commands are given in proper sequence. Please become thoroughly familiar with all the commands and follow the examples given in this chapter. It's important to follow the handshake conventions exactly for maximum performance.

With the exception of READ and WRITE, burst commands do not translate from logical to physical track and sector. All track and sector parameters refer to physical locations (see Chapter 10). Burst sector READ and WRITE commands provide a flag to enable logical to physical translation. If the flag is set, the drive does the translation and the default logical number of bytes per sector (256) is transferred instead of the physical number of bytes per sector (512).

CMD 1—READ

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	L	E	X	S	0	0	0	N
03	DESTINATION TRACK							
04	DESTINATION SECTOR							
05	NUMBER OF SECTORS							
06	NEXT TRACK (OPTIONAL)							

RANGE: All values are determined by the particular disk format and format of translation table.

SWITCHES: L—logical flag (1 = do logical to physical translation)
E—ignore error (1 = ignore)
S—side select
N—drive number

PROTOCOL: Burst handshake

CONVENTIONS: Before you can READ or WRITE to a diskette, it must be logged-in using either the INQUIRE DISK or QUERY DISK FORMAT command (both are described later). This must be done once each time you change diskettes.

OUTPUT: One burst status byte, followed by burst data, is sent for each sector transfered. An error prevents data from being sent unless the E bit is set.

CMD 2—WRITE

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	L	E	X	S	0	0	1	N
03	DESTINATION TRACK							
04	DESTINATION SECTOR							
05	NUMBER OF SECTORS							
06	NEXT TRACK (OPTIONAL)							

RANGE: All values are determined by the particular disk format and format of translation table.

SWITCHES: L—logical flag (1 = do logical to physical translation)
E—ignore error (1 = ignore)
S—side select
N—drive number

PROTOCOL: Burst data to the drive, then host must perform the following: fast serial input, pull the clock low and wait for the burst status byte, pull clock high, go output for multi-sector transfers and continue.

CONVENTIONS: Before you can READ or WRITE to a diskette, it must be logged-in using either the INQUIRE DISK or QUERY DISK FORMAT command (both are described later). This must be done once each time you change diskettes.

INPUT: Host must transfer burst data.

OUTPUT: One burst status byte following each WRITE operation.

CMD 3—INQUIRE DISK

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	S	0	1	0	N

SWITCHES: N—drive number

PROTOCOL: Burst handshake

OUTPUT: One burst status byte following each INQUIRE DISK operation.

CMD 4—FORMAT

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	M	X	X	X	0	1	1	N
03	SECTOR SIZE			*(OPTIONAL, DEF-02,512 BYTE SECTORS)				
04	LAST TRACK NUMBER			(OPTIONAL, DEF-79)				
05	NUMBER OF SECTORS			**(OPTIONAL, DEF DEPENDS ON BYTE 03)				
06	STARTING TRACK			(OPTIONAL, DEF-0)				
07	FILL BYTE			(OPTIONAL, DEF-$E5)				
08	STARTING SECTOR			(OPTIONAL, DEF-1)				

*01—256 BYTE SECTORS
02—512 BYTE SECTORS
03—1024 BYTE SECTORS

**16—256 BYTE SECTORS
10—512 BYTE SECTORS
5—1024 BYTE SECTORS

SWITCHES: M—MODE (1 = will accept BYTES 03 - 08, 0 = will format, create directory and BAM)
N—drive number

PROTOCOL: Conventional

CONVENTIONS: CMD 4 must be followed with CMD 3 or CMD 6 once to log the disk in.

OUTPUT: None. Status will be updated within the drive.

CMD 6—QUERY DISK FORMAT

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	F	X	T	S	1	0	1	N
03	OFFSET (OPTIONAL F-BIT SET)							

SWITCHES: F—force flag (F = 1 steps the head with the offset specified in byte 03)
T—sector table (T = 1, send sector table)
N—drive number
X—don't care
S—side select

PROTOCOL: Burst handshake

CONVENTIONS: Determines the diskette format on any particular track. Also logs non-standard diskettes (i.e. minimum sector addresses other than one).

OUTPUT: *burst status byte (no bytes will follow if there is an error)
**burst status byte (no bytes will follow if there was an error in compiling MFM format information)
number of sectors (the number of sectors on a particular track)
logical track (the logical track number found in the disk header)
minimum sector (the logical sector with the lowest value address)
maximum sector (the logical sector with the highest value address)
interleave (always returns 1)
sector table (with T bit set, sector table is sent)
*status from track offset zero
**if F bit is set, status is from offset track

CMD 7—INQUIRE STATUS

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	W	C	M	0	1	1	0	N
03	NEW STATUS (W-BIT CLEAR)							
04	NEW ORA MASK (M-BIT SET)							
05	NEW AND MASK (M-BIT SET)							

SWITCHES: W—write switch (0 = write)
M—write AND/OR mask (04 new OR mask (M-bit set), 05 new AND mask (M-bit set))
C—change (C = 1 and W = 0—log in disk C = 1 and W = 1—return whether disk was logged, i.e. $B error or old status)
N—drive number
X—don't care

PROTOCOL: Burst handshake (W = 1), conventional (W = 0)

CONVENTIONS: This is a method of reading or writing current status, and changing the status mask value.

OUTPUT: None (W = 0), Burst status byte (W = 1)

CMD 8—DUMP TRACK CACHE BUFFER

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	F	S	X	1	1	1	0	1
03	PHYSICAL TRACK							

SWITCHES: X—don't care
S—side select
F—(1 = write even if not "dirty")

CHGUTL UTILITY

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	1	1	1	1	0
03	UTILITY COMMANDS: 'B', 'S', 'R', 'T', 'V', #DEV, "MR", "MW"							
04	COMMAND PARAMETER							

SWITCHES: X—don't care

UTILITY COMMANDS: 'B'—serial bus mode
'S'—DOS sector interleave
'R'—DOS retries
'T'—ROM signature analysis
'V'—verify select
#DEV—device #

Note: Byte 02 is equivalent to '>'.

EXAMPLES: "U0>B1" = Fast Serial, "U0>B0" = Slow Serial
"U0>S" + CHR$ (SECTOR INTERLEAVE)
"U0>R" + CHR$ (RETRIES)
"U0>T" (If the ROM signature failed, the activity LED blinks 4 times)
"U0>V0" = Disk Verify ON, "U0>V1" = Disk Verify OFF
"U0>" + CHR$ (#DEV), where #DEV = 4 – 30
"U0>MR" + CHR$ (>memory address) + (# of pages)*
"U0>MW" + CHR$ (>memory address) + (# of pages)*

*Burst memory read and memory write use standard burst protocol (without status byte).

FASTLOAD UTILITY

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	P	X	X	1	1	1	1	1
03	FILE NAME							

SWITCHES: P—sequential file bit (P = 1, does not have to be a program file)
X—don't care

PROTOCOL: Burst handshake

OUTPUT: Burst status byte preceding each sector transfered.

STATUS IS AS FOLLOWS: *0000000*X—OK
* *00000010*—file not found
** *000*11111—EOI

*Values between 3 and 15 should be considered a file read error.
**The byte following the EOI status byte is the number of data bytes to follow.

STATUS BYTE BREAKDOWN

BIT 7	6	5	4	3	2	1	0
MODE	DN	SECTOR SIZE		[	CONTROLLER STATUS		]

*MODE—1 = Alien Disk Format (non-default physical format, or default physical format without Directory and BAM information)
0 = Resident Disk Format (default physical format with Directory and BAM information)
DN—DRIVE NUMBER

*Resident Disk Format is based on whether required information is present in BAM/DIRECTORY track.

SECTOR SIZE

00 . . . . . 128 BYTE SECTORS (NOT SUPPORTED)
01 . . . . . 256 BYTE SECTORS
10 . . . . . 512 BYTE SECTORS (DEFAULT)
11 . . . . . 1024 BYTE SECTORS

CONTROLLER STATUS

000X . . . . . OK
0010 . . . . . CAN'T FIND HEADER BLOCK
0011 . . . . . NO ADDRESS MARK
0100 . . . . . DATA BLOCK NOT PRESENT
0101 . . . . . DATA CRC ERROR
0110 . . . . . FORMAT ERROR
0111 . . . . . VERIFY ERROR
1000 . . . . . WRITE PROTECT ERROR
1001 . . . . . HEADER BLOCK CRC ERROR
1010 . . . . . WRITE PROTECTED

1011 ..... DISK CHANGE
1100 ..... DISK FORMAT NOT LOGICAL
1101 ..... RESERVED
1110 ..... SYNTAX ERROR
1111 ..... NO DRIVE PRESENT

BURST TRANSFER PROTOCOL

Before using the following burst transfer routines, you must determine whether or not the peripheral is a fast device. The Fast Serial (byte mode) protocol makes that determination internally when you include a query routine (send-cmd-string;). This routine addresses the peripheral as a listener and thereby determines its speed.

BURST READ

```
send-cmd-string;                    (*determine speed*)
if device-fast then
    serial-in;                      (*turn 8520 to input*)
    repeat                          (*repeat for all sectors*)
        read-error;                 (*retrieve error byte*)
toggle clock;                       (*wait for status*)
repeat
wait-byte;                          (*wait for byte*)
if status = ok then
    toggle-clock;                   (*start double buffer*)
    repeat
        wait-byte;                  (*get data*)
        toggle-clock;               (* start next*)
        store-byte;                 (*save data*)
    until end-of-sector;
until no-more-sectors;
    set-clock-high;                 (*release clock line*)
else
    read-1541;                      (*send unit read*)
```

BURST WRITE

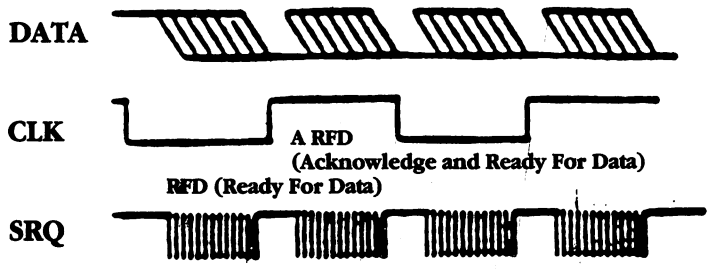
```
send-cmd-string;                    (*determine speed*)
if device-fast then
    repeat                          (*repeat for multi-sector*)
        serial-out;                 (*serial port out*)
            repeat                  (*repeat for sector-size*)
                send-byte;          (*send byte*)
            until last-byte;        (*last byte ?*)
        serial-in;                  (*serial port in*)
        clock-low;                  (*ready for status*)
        read-err;                   (*controller error ?*)
        clock-high;                 (*restore clock*)
    until last-sector;              (*until last sector*)
else
    write-1581;                     (*unit write*)
```

EXPLANATION OF PROCEDURES

send-cmd-string	sends one byte of the command to determine whether the drive is fast or slow.
toggle-clock	changes the state of the clock line.
clock-hi	changes the state of the clock to logic 1.
clock-lo	changes the state of the clock to logic Ø.
wait-byte	polls the 8520 for a byte ready.
read-error	calls toggle-clock and wait-byte, then returns to the main if there are no errors.
store-data	stores the data in a particular memory location.
last-byte	depending on sector size, will increment and compare value to sector size.
last-sector	decrements the number of sector transfers requested and stops when done.
serial-in	sets the 8520 serial port and driver circuit to input mode.
read-err	calls wait-byte and evaluates the status of the previous controller job.
serial-out	sets the 8520 serial port and driver circuit to output mode.
send-byte	sends a byte of data to the 1581.
read-1581	sends a typical unit read to a 1581.
write-1581	sends a typical unit write to a 1581.

The figure below shows the burst transfer protocol. It is a stat-dependent protocol (simple and fast). As the clock line is toggled, a byte of data is sent. Burst protocol is divided into three parts:

1. Send Command: send string using existing kernal routines.
2. Query: determine whether the peripheral is fast.
3. Handshake Code: follow handshake conventions.



NOTE

An example of using the burst routines is on the test/demo diskette. Print those files and use them as references for creating your own programs that use burst protocol.

CHAPTER 10
1581 INTERNAL OPERATION

This chapter describes some of the internal operations of the 1581—how things work on the 'other side of the fence' of the host computer. Experienced programmers may find this information useful. The information learned in previous chapters (especially Chapter 7) combined with that presented in this chapter provides a wide realm of possibilities for the creative and persistent programmer. Please be reminded, however, of the NOTE presented at the beginning of Chapter 7 regarding future compatibility.

Logical versus Physical Disk Format

All DOS operations of the 1581 are done in 256 byte blocks. These blocks appear as individually numbered sectors on the disk. By going through the DOS interface via the commands outlined in this manual, the logical disk format is as follows:

single sided
80 tracks (track 1 through track 80)
40 256 byte sectors per track (sector 0 through sector 39)

Internally, however, the 1581 has a different view of things. The disk is actually formatted as follows:

double sided (side 0 and side 1)
80 tracks per side (track 0 through track 79)
10 512 byte sectors per track (sector 1 through sector 10)

That is the physical disk format created whenever a HEADER (NEW) command is sent to the 1581 from the host computer. The physical format is different from the logical so that more data can be squeezed onto each disk.

All commands sent from the host computer are parsed through the DOS (except the BURST commands) and refer to the logical format. Software inside the 1581 automatically takes care of the logical to physical translations necessary to retrieve the data properly from the disk.

Track Cache Buffer

One of the improvements in the 1581 design over previous Commodore disk drives which makes the device more efficient is the Track

Cache Buffer. This buffer is located from $0C00 through $1FFF. All disk accesses involve an entire physical track at a time. If a single sector is needed from a particular track, the entire track is read into RAM. Consequently, any more requests for sectors from the same track require only a RAM data transfer, rather than a search of the track on the disk again. Sector writes are also speeded up considerably, since each consecutive write to sectors on the same track requires only a RAM to RAM data transfer. After data is written into the Track Cache Buffer, it is not written to the disk until one of the following occurs: 1) a request is made for access of a sector on a different track, 2) a 'dump track buffer' command is issued, or 3) after 250 milliseconds of no serial bus activity.

Controller Job Queue

The software in the 1581's ROM can be broken down into two major components—the DOS and the Controller. The DOS (Disk Operating System) is the software interface between the host computer and the 1581 system. The DOS keeps track of the file management details necessary to create, modify and delete files. It monitors the amount of free space left on a disk, and keeps track of the file names in the directory. It remembers where each file starts, and ensures that none of them overlap. When the host sends commands, the DOS checks to make sure that the syntax and the parameters are valid. The DOS is very complex, and its code occupies the majority of the ROM.

The Controller, on the other hand, knows nothing about the concept of files. The Controller deals only in the physical world of the disk itself. It is responsible for reading and writing each of the individual sectors on the disk.

The DOS and Controller communicate to each other primarily through a 'mailbox,' known as the Job Queue. Because the DOS deals more in the abstract, or logical, world, it does not access the physical disk itself. If the DOS requires access to the disk, it must ask the controller to do it. The DOS places a Controller Command Code into the Job Queue and then waits for the requested operation to be completed. The Controller polls the Job Queue every 10 milliseconds (the polling rate is determined by timer B of the 8520) looking for something to do. If there is a job in the queue, the Controller executes the job and returns a status byte to the DOS. To speed things up, the polling of the job queue can be bypassed by a direct jump to JSTROBECONTROLLER at $FF54 (refer to the section describing the Vectored Jump Table). A call to JSTROBECONTROLLER requires the command in the accumulator and the job queue offset in the x register.

Most of the tasks that the Controller can be asked to perform require parameters. The parameters are also placed in a 'mailbox' location for the Controller to access (prior to placing the command byte into the queue, of course). These parameters can be either the physical or the logical parameters, depending on what the command expects to see. Logical parameters are placed in HDRS as two consecutive bytes of track, sector. Physical parameters are also placed in HDRS, but in addition the side must be specified (in SIDS). Commands that require logical parameters must translate them into physical parameters at some point in their execution. The logical parameters are translated into physical parameters by a vectored routine called TRANSTS, and are written into HDRS2 and SIDS. The DOS will always pass logical parameters, and uses only the Controller commands that expect to see logical parameters. However, the Controller is also allowed to put jobs in its own queue, so it needs to pass physical parameters as well as logical.

Mailbox Locations of the Controller

name	address	purpose
JOBS	$0002	JOB queue for Controller commands (JOBS 0-8).
	.	Each JOB uses 1 byte. The last 2 locations
	.	($09,$0A) are reserved for BAM jobs only!
	$000A	
HDRS	$000B	Logical or physical track, sector for each of
	.	the jobs in the JOB queue (2 bytes per job).
	.	
	$001C	
HDRS2	$01BC	Translated (physical) track, sector for each
	.	of the jobs in the JOB queue (2 bytes per job).
	.	
	$01CD	
SIDS	$01CE	Physical side for each of the jobs
	.	in the job queue (1 byte per job).
	.	
	$01D6	
CACHE	$008B	Pointer to BUFFCACHE below.
CACHEOFF	$009F	Offset into the track cache buffer.
	.	(1 byte per job).
	$00A7	
BUFFCACHE	$0C00	20 pages for track cache.

Associated with each of the nine Job Queue locations (Jobs 0 through 8) are nine 256 byte buffers (buffers 0-8), beginning at location $0300. Data passed from/to a particular Job during its execu-

tion is located in the buffer which corresponds to the position of the Controller Job Code in the Job Queue. For example, if the Controller code in Job Queue position 2 (at location $0004) requests that a logical sector of data be read, the data is put into Buffer 2 (at location $0500).

Table 1 is a list of Controller Job Codes that can be put into the Job Queue. Table 2 is a list of the codes that are returned by the Controller once the job has been executed. The return code is placed into the Job Queue in the same memory location that contained the Controller Job Code. Consequently, the procedure to use the Controller is the following:

1) Write any parameters needed by the Job into the parameter variables (HDSRS, HDRS2, SIDS).
2) Write the Controller Code into the Job Queue (JOBS).
3) Wait for the job to be completed by simply polling the location in JOBS where the Job code was put, and waiting for it to change (bit 7 will be reset to 0).

TABLE 1: Controller JOB Command Codes

name	code	description
READ_DV	$80	Reads a particular logical sector into the job queue buffer (only if the disk has not been changed). If the desired sector is already in the track cache buffer, then no disk activity is required (the data is merely transferred from the track cache memory to the job queue buffer memory). If the desired sector is not in the track cache, then the current track cache is dumped to disk (only if it has been modified), the desired track is read into the track cache, and finally the particular sector's data is transferred from the track cache memory to the job queue buffer.

WRTSD_DV	$90	Writes the job queue's buffer data to a particular logical track, sector. If the same track is already in the track cache, then this involves only transferring the job queue buffer data to the track cache buffer. If a different track's data is in the track cache, then it must first be dumped to the disk (only if it was modified), the desired track read into the track cache buffer, and finally the job queue buffer's data transferred to the track cache.
WRTVER_DV	$A0	Verifies the track cache buffer's data against the specified logical track's data.
SEEKHD_DV	$B0	Logs in a disk by reading information from the first header encountered on the disk into RAM so that it can be used by the DOS. The track cache buffer is not updated.
SEEKPHD_DV	$B8	Seeks to a particular logical track, sector. The track cache is not updated.
RESTORE_DV	$C0	Restores the read/write head to track 0 ('bump').
JUMPC_DV	$D0	Executes the code in the corresponding job queue buffer.
EXBUF_DV	$E0	Executes the code in the corresponding job queue buffer after the motor is up to speed and the head is on track.
RESET_DV	$82	Resets the disk controller and associated variables.
MOTON_DV	$84	Turns on the spindle motor (overlays a $01 in the Job Queue after the spin-up sequence is complete).
MOTOFF_DV	$86	Turns the spindle motor off after the spin-down sequence is complete.
MOTONI_DV	$88	Turns the spindle motor on immediately.
MOTOFFI_DV	$8A	Turns the spindle motor off immediately.
SEEK_DV	$8C	Seeks to a particular physical track (cylinder). The current physical track position should be put in the track parameter of HDRS.
FORMAT_DV	$8E	Formats one physical track (one half of a cylinder). The head must be placed physically over the proper cylinder, and the head electronics must be selected for the side desired.
DISKIN_DV	$92	Determines if there is a disk inserted in the drive.
LEDACTON_DV	$94	Turns on the activity LED.
LEDACTOFF_DV	$96	Turns off the activity LED.
ERRLEDON_DV	$98	Enables error LED blinking.
ERRLEDOFF_DV	$9A	Disables error LED blinking.
SIDE_DV	$9C	Sets up the side select electronics to the value specified (in SIDS).

BUFMOVE_DV	$9E	Moves data between the job queue buffer and the track cache buffer. The track parameter in the job queue denotes the position in the track cache buffer to transfer to/from. The sector parameter denotes the following: Bit 7 : Direction (1 = to track cache buffer) Bit 6 : Mark Flag (set/clear the 'track cache modified' flag) Bit 5 : Transfer (1 = do the transfer) Bits 4-0 : # of 256 byte blocks to transfer With bit 7 set, the corresponding physical track position in the job queue (HDRS2) must be updated for the purpose of telling the controller what physical track the track cache buffer belongs to. In addition the side var (SIDS) must also be updated.
TRKWRT_DV	$A2	Dumps the track cache buffer to the disk (only if the track cache modified flag is set).
SP_READ	$A4	Reads the specified physical sector directly into RAM starting at #0 ($0300). It does not use the track cache buffer. The sector is always read from the disk regardless of the current contents of the track cache.
SP_WRITE	$A6	Writes to the specified physical sector directly. It does not use the track cache. Data to be written starts at buffer #0 ($0300).
PSEEK_DV	$A8	Seeks to the specified physical track.
TREAD_DV	$AA	Reads logical address without transferring to the job queue buffer.
TWRT_DV	$AC	Writes a logical address without transferring from the job queue buffer.
TPREAD_DV	$B2	Reads a physical address without transferring to the job queue buffer.
TPWRT_DV	$B4	Writes a physical address without transferring from the job queue buffer.
DETWP_DV	$B6	Checks if the disk inserted is write protected. Returns $00 if disk is not protected, else $08.
FORMATDK_DV	$F0	Formats the disk with the default physical format.

TABLE 2: Controller JOB Return Codes

name	code	description
OK_DV	$0x	No error.
MISHD_DV_ER	$02	Can't find header block.
NOADAM_DV_ER	$03	No address mark detected.
MISDBLK_DV_ER	$04	Data block not present.
CRCDBLK_DV_ER	$05	CRC error encountered in data block.
FMT_DV_ER	$06	Format error.
VERERR_DV_ER	$07	Verify error.
WRTPR_DV_ER	$08	Attempt to write to a write protected disk.
CRCHD_DV_ER	$09	CRC error encountered in header block.
	$0A	Reserved.
DSKCHG_DV_W	$0B	Disk was changed / disk ID mismatch.
DSKNTLOG_DV_ER	$0C	Disk format not logical.
CONTROLLER_ER	$0D	Floppy disk controller IC error.
SYNTAX_DV-ER	$0E	Syntax error. Invalid job number.
NODSKPRS_DV_ER	$0F	No disk is present in the drive.

Here is an example BASIC program which will display the data on any sector of the 1581 disk. It puts a READ SECTOR job directly into the Controller Job Queue using MEMORY-WRITE, and then reads the sector data directly from the Job Queue Buffer using MEMORY-READ.

```
0010  OPEN 1,8,15 : REM OPEN COMMAND CHANNEL TO 1581
0020  OPEN 2,8,2,"#" : REM OPEN BUFFER #0 OF THE 1581 (AT $0300
        OF 1581)
0030  INPUT "TRACK, SECTOR TO READ";T,S
0040  REM WRITE TRACK, SECTOR PARAMETERS TO HDRS IN 1581 RAM
        (AT $000B)
0050  PRINT#1,"M-W"+CHR$(11)+CHR$(0)+CHR$(2)+CHR$(T)
        +CHR$(S)
0060  REM PUT THE READ SECTOR ($80) COMMAND INTO THE JOB
        QUEUE
0070  PRINT#1, "M-W"+CHR$#(2)+CHR$(0)+CHR$(1)+CHR$(128)
0080  REM READ BACK THE JOB QUEUE WHERE THE COMMAND WAS
        JUST WRITTEN,
0090  REM WAITING FOR THE STATUS TO BE WRITTEN INTO IT.
0100  PRINT#1,"M-R"+CHR$(2)+CHR$(0)+CHR$(1)
0110  GET#1,A$ : IF ASC(A$) > 127 THEN 100
0120  PRINT "STATUS RETURNED = ";ASC(A$)
0130  REM READ THE TRANSLATED TRACK, SECTOR VALUES FROM
        HDRS2
0130  PRINT#1, "M-R"+CHR$(188)+CHR$(01)+CHR$(02)
0140  GET#1,A$ : PT=ASC(A$)
0150  GET#1,A$ : PS=ASC(A$)
```

```
0160  REM READ THE TRANSLATED SIDE VALUE
0170  PRINT#1, "M-R"+CHR$(239)+CHR$(0)+CHR$(1)
0180  GET#1,A$ : SIDE=ASC(A$)
0190  PRINT "TRANSLATED TRACK=";PT;" SECTOR=";PS;" SIDE=";SIDE
0200  PRINT
0210  REM READ THE 256 BYTES OF DATA FROM 1581 JOB QUEUE
          BUFFER INTO CPU RAM
0220  PRINT#1, "M-R"+CHR$(0)+CHR$(3)+CHR$(0)
0230  REM LIST THE SECTOR DATA
0240  PRINT"     ";
0245  PRINT"0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F"
0250  PRINT"-------";
0255  PRINT"------------------------------------------------------------"
0260  FOR X=0 TO 15
0270  PRINT RIGHT$(HEX$(X),1);TAB(6);
0280  FOR Y=1 TO 16
0290  GET#1,A$
0300  PRINTUSING "####";RIGHT$(HEX$(ASC(A$)),2);
0310  NEXT Y
0320  PRINT
0330  NEXT X
```

Vectored Jump Table

Each of the DOS commands that can be sent to the 1581 via the serial bus are vectored through indirect jumps in the ROM. The indirect vectors are located in RAM, so the user can change these vectors for the purpose of providing a different routine, or massaging data before passing control to the original routine. Each of these vectors and their locations are listed in the table 3 :

TABLE 3: Indirect Vector Jump Table

name	location	description
JIDLE	$FF00	Main idle loop. When a Controller command is completed the IDLE routine is executed. It first checks to see if there are any more jobs pending in the Job Queue. If so, it executes them. If not, it sits in the idle loop waiting for something to happen, such as another job being put into the queue, ATN line going low, disk inserted or removed, etc.

JIRQ	$FF03	Interrupt routine. Interrupts normally occur from the following sources — ATN line going low, Fast Serial Byte is shifted in, timer time out, execution of a BRK instruction.
JNMI	$FF06	Does a 'soft' reset (UI command). Default vectors and variables are restored. Searches for "COPYRIGHT CBM 86" USR type file to boot. No RAM check or ROM checksum is done. Device # switches are read.
JVERDIR	$FF09	VALIDATE command (collect).
JINTDRV	$FF0C	INITIALIZE command.
JPART	$FF0F	Routine to create or switch partitions.
JMEM	$FF12	Memory Read/Memory Write (M-R,M-W) commands.
JBLOCK	$FF15	Performs all BLOCK commands, such as ALLOCATE, FREE, READ, WRITE, EXECUTE, POINTER.
JUSER	$FF18	USER command.
JRECORD	$FF1B	RECORD command for relative file positioning.
JUTLODR	$FF1E	Utility loader command (&).
JDSKCPY	$FF21	Copy command.
JRENAME	$FF24	Rename command.
JSCRTCH	$FF27	Scratch command.
JNEW	$FF2A	New/Format command.
ERROR	$FF2D	Controller error handler routine.
JATNSRV	$FF30	Serial Bus attention (ATN) server.
JTALK	$FF33	Serial Bus talk routine.
JLISTEN	$FF36	Serial Bus listen routine.
JLCC	$FF39	Controller routine.
JTRANS_TS	$FF3C	Logical to physical sector translation routine.
CMDERR	$FF3F	DOS error handler routine.
JSTROBE_ CONTROLLER	$FF54	Direct Controller call.
JCBMBOOT	$FF57	CBM autoloader routine.
JCBMBOOTRTN	$FF5A	Return from CBM autoloader with autoloader disabled.
JSIGNATURE	$FF5D	Signature analysis routine.
JDEJAVU	$FF60	Switch for autoloader boot on INITIALIZE or BURST INQUIRE/QUERY. Enter with carry set to enable autoloader, carry to clear disable it.
JSPINOUT	$FF63	SPIN, SPOUT. Sets up fast serial direction as input or output. Carry set to do SPOUT, cleared to do SPINP.
JALLOCBUFF	$FF66	Allocates RAM buffers. Call with buffer # in reg A (1 = buffer 0,2 = buffer 1...).

APPENDIX A
CHANGING THE DEVICE NUMBER

Two switches on the back of the 1581 enable you to change the device # of the drive. You can use a screwdriver, pen, or any other small tool to set the switches. The following table shows the settings required for each device number:

Left	Right	Device #
UP	UP	8
DOWN	UP	9
UP	DOWN	10
DOWN	DOWN	11

Another way to temporarily change the device number of a disk drive is via a program. When power is first turned on, the drive reads an I/O location whose value is controlled by the two switches on its circuit board, and writes the device number it reads there into memory locations 119 and 120. Any time thereafter, you may write over that device number with a new one, which will be effective until it is changed again, or the 1581 is reset.

FORMAT FOR TEMPORARILY CHANGING THE DISK DEVICE NUMBER:

PRINT#15,"U0>" + CHR$(n)

Where n = 8 to 30

EXAMPLE:

Here is a program that sets any device number:

```
5  INPUT "OLD DEVICE NUMBER"; ODV
10 INPUT "NEW DEVICE NUMBER"; DV
20 IF DV<8 or DV>30 then 10
30 OPEN 15,ODV,15, "U0>"+CHR$(DV): CLOSE 15
```

NOTE

If you will be using two disk drives, and want to temporarily change the device number of one, you will need to run the above program with the disk drive whose device number is not to be changed turned off. After the program has been run, you may turn that drive back on. If you need to connect more than two drives at once, you will need to use the hardware method of changing device numbers.

APPENDIX B
DOS ERROR MESSAGES

Many commercial program diskettes are intentionally created with one or more of the following errors, to keep programs from being improperly duplicated. If a disk error occurs while you are making a security copy of a commercial program diskette, check the program's manual. If its copyright statement does not permit purchasers to copy the program for their own use, you may not be able to duplicate the diskette. In some such cases, a safety spare copy of the program diskette is available from your dealer or directly from the company for a reasonable fee.

00: OK (not an error)
This is the message that usually appears when the error channel is checked. It means there is no current error in the disk unit.

01: FILES SCRATCHED (not an error)
This is the message that appears when the error channel is checked after using the SCRATCH command. The track number tells how many files were erased.

02: PARTITION SELECTED (not an error)
The disk partition requested has been selected.

NOTE

If any other error message numbers less than 20 ever appear, they may be ignored. All true errors have numbers of 20 or more.

20: READ ERROR (block header not found)
The disk controller is unable to locate the header of the requested data block. Caused by an illegal block or a header that has been destroyed. Usually unrecoverable.

21: READ ERROR (drive not ready)
The disk controller is unable to detect a sync mark on the desired track. Caused by misalignment, or a diskette that is absent, unformatted or improperly seated. Can also indicate hardware failure. Unless caused by one of the above simple causes, this error is usually unrecoverable.

22: READ ERROR (data block not found)
The disk controller has been requested to read or verify a data block that was not properly written. Occurs in conjunction with BLOCK commands and indicates an illegal track and/or sector request.

23: READ ERROR (CRC error in data block)
There is an error in the data. The sector has been read into disk memory, but its CRC is wrong.

24: READ ERROR (bad sector header)
The data or header has been read into disk memory, but a hardware error has been created by an invalid bit pattern in the data byte.

25: WRITE ERROR (write-verify error)
The controller has detected a mismatch between the data written to diskette and the same data in disk memory. May mean the diskette is faulty. If so, try another. Use only high-quality diskettes from reputable makers.

26: WRITE PROTECT ON
The controller has been requested to write a data block while the write-protect sensor is uncovered. Usually caused by writing to a diskette whose write protect notch is pushed back to expose the hole to prevent changing the diskette's contents.

27: READ ERROR (CRC error in header)
The controller detected an error in the header bytes of the requested data block. The block was not read into disk memory.

30: SYNTAX ERROR (general syntax)
The DOS cannot interpret the command sent to the command channel. Typically, this is caused by an illegal number of file names or an illegal pattern. Check your typing and try again.

31: SYNTAX ERROR (invalid command)
The DOS does not recognize the command. It must begin with the first character sent. Check your typing and try again.

32: SYNTAX ERROR (long line)
The command sent is longer than 58 characters. Use abbreviated disk commands.

33: SYNTAX ERROR (invalid file name)
Pattern matching characters cannot be used in the SAVE command or when Opening files for the purpose of Writing new data. Spell out the file name.

34: SYNTAX ERROR (no file given)
The file name was left out of a command or the DOS does not recognize it as such. Typically, a colon (:) has been omitted. Try again.

39: SYNTAX ERROR (invalid command)
The DOS does not recognize a command sent to the command channel (secondary address 15). Check your typing and try again.

50: RECORD NOT PRESENT
The requested record number has not been created yet. This is not an error in a new relative file or one that is being intentionally expanded. It results from reading past the last existing record, or positioning to a non-existent record number with the Record# command.

51: OVERFLOW IN RECORD
The data to be written in the current record exceeds the record size. The excess has been truncated (cut off). Be sure to include all special characters (such as carriage returns) in calculating record sizes.

52: FILE TOO LARGE
There isn't room left on the diskette to create the requested relative record. To avoid this error, create the last record number that will be needed as you first create the file. If the file is too large for the diskette, either split it into two files on two diskettes, or use abbreviations in the data to allow shorter records.

60: WRITE FILE OPEN
A write file that has not been closed is being reopened for reading. This file must be immediately rescued, as described in BASIC Hint #2 in Chapter 2, or it will become a splat (improperly closed) file and probably be lost.

61: FILE NOT OPEN
A file is being accessed that has not been opened by the DOS. In some such cases no error message is generated. Rather the request is simply ignored.

62: FILE NOT FOUND
The requested file does not exist on the indicated drive. Check your spelling and try again.

63: FILE EXISTS
A file with the same name as has been requested for a new file already exists on the diskette. Duplicate file names are not allowed. Select another name.

64: FILE TYPE MISMATCH
The requested file access is not possible using files of the type named. Reread the chapter covering that file type.

65: NO BLOCK
Occurs in conjunction with B-A. The sector you tried to allocate is already allocated. The track and sector numbers returned are the next higher track and sector available. If the track number returned is 0, all remaining sectors are full. If the diskette is not full yet, try a lower track and sector.

66: ILLEGAL TRACK AND SECTOR
The DOS has attempted to access a track or sector which does not exist. May indicate a faulty link pointer in a data block.

67: ILLEGAL SYSTEM T OR S
This special error message indicates an illegal system track or block.

70: NO CHANNEL (available)
The requested channel is not available or all channels are in use. A maximum of three sequential files or one relative file plus one sequential file may be opened at one time, plus the command channel. Do not omit the drive number in a sequential OPEN command, or only two sequential files can be used. Close all files as soon as you no longer need them.

71: DIRECTORY ERROR
The BAM (Block Availability Map) on the diskette does not match the copy in disk memory. To correct, Initialize the diskette.

72: DISK FULL
Either the diskette or its directory is full. DISK FULL is sent when two blocks are still available, allowing the current file to be closed. If you get this message and the directory shows any

blocks left, you have too many separate files in your directory, and will need to combine some, delete any that are no longer needed, or copy some to another diskette.

74: DRIVE NOT READY
An attempt has been made to access the 1581 single disk without a formatted diskette in place. Blank diskettes cannot be used until they have been formatted.

75: FORMAT ERROR

76: CONTROLLER ERROR
The floppy disk controller IC (WD177x) is not functioning properly.

77: SELECTED PARTITION ILLEGAL
An attempt has been made to select a partition that does not meet the criteria of a directory partition.

APPENDIX C
DOS DISKETTE FORMAT

The DIRECTORY and BAM are located on logical track 40. The following is the structure:

DIRECTORY HEADER (Track 40 sector 0)

BYTE	DEFINITION
00-01	Track and Sector of first DIRECTORY block.
02	Disk Version Number
03	$00
04-21	Disk Name
22-23	Disk Id
24	$A0
25	DOS Version Number
26	Disk Version Number
27-28	A0h
29-255	$00

BAM for Logical Tracks 1-40 — BAM1 (Track 40 Sector 1)

BYTE	DEFINITION
00-01	Track and Sector of next BAM block
02	Version Number
03	Compliment Version Number
04-05	Disk Id
06	I/O-Byte (bit7-verify on/off, bit6-check header CRC on/off)
07	Auto Loader Flag*
08-15	Reserved for future use
16-255	BAM image for logical tracks 1-40 (6 bytes per track**)

*When the drive is reset it will hold off the serial bus and look for a file called "COPYRIGHT CBM 86" of file type USR. It will load and execute this file. The file must have the following structure: The first two bytes of the file must contain the low and high load addresses respectively. The third byte is the amount of characters to follow. In addition a trailing checksum byte must be included. The load address is the execution address. The BAM contains a flag byte which will allow auto execution with an Initialize, Burst, Inquire, and Burst Query commands.

BAM for Logical Tracks 41-80 — BAM2 (Track 40 Sector 2)

BYTE	DEFINITION
00	00
01	$FF
02	Version Number (copy)
03	Compliment Version Number (copy)
04-05	Disk Id (copy)
06	I/O byte (copy)
07	Auto Loader Flag (copy)
08-15	Reserved for future use
16-255	BAM image for logical tracks 41-80 (6 bytes per track**)

****Format of 6 BAM bytes for each track**

BYTE OFFSET	DEFINITION
0	Number of free sectors on track
1	MSB — flag for sector 7, LSB — flag for sector 0
2	MSB — flag for sector 15, LSB — flag for sector 8
3	MSB — flag for sector 23, LSB — flag for sector 16
4	MSB — flag for sector 31, LSB — flag for sector 24
5	MSB — flag for sector 39, LSB — flag for sector 32

DIRECTORY FILE FORMAT
Track 40, Sectors 3-39

BYTE	DEFINITION
0,1	Track and sector of next directory block.
2-31	File entry 1*
34-63	File entry 2*
66-95	File entry 3*
98-127	File entry 4*
130-159	File entry 5*
162-191	File entry 6*
194-223	File entry 7*
226-255	File entry 8*

*STRUCTURE OF EACH INDIVIDUAL DIRECTORY ENTRY

BYTE OFFSET	CONTENTS	DEFINITION
0	128 + type	File type OR'ed with $80 to indicate properly closed file. (if OR'ed with $C0 instead, file is locked.) TYPES: 0 = DELeted 1 = SEQuential 2 = PROGram 3 = USER 4 = RELative 5 = CBM
1-2		Track and sector of first data block.
3-18		File name padded with shifted spaces.
19-20		Relative file only: track and sector of the super side sector block.
21		Relative file only: record length.
22-25		Unused.
26-27		Track and sector of replacement file during an @SAVE or @OPEN.
28-29		Number of blocks in file: stored as a two-byte integer, in low-byte, high-byte order.

PROGRAM FILE FORMAT

BYTE	DEFINITION
FIRST SECTOR	
0,1	Track and sector of next block in program file 1.
2,3	Load address of the program.
4-255	Next 252 bytes of program information stored as in computer. memory (with key words tokenized).
REMAINING FULL SECTORS	
0,1	Track and sector of next block in program file 1.
2-255	Next 254 bytes of program information stored as in computer memory (with key words tokenized).
FINAL SECTOR	
0,1	Null ($00), followed by number of valid data bytes in sector.
2-???	Last bytes of the program information, stored as in computer memory (with key words tokenized). The end of a BASIC file is marked by three zero bytes in a row. Any remaining bytes in the sector are garbage and may be ignored.

SEQUENTIAL FILE FORMAT

BYTE	DEFINITION
ALL BUT FINAL SECTOR	
0-1	Track and sector of next sequential data block.
2-255	254 bytes of data.
FINAL SECTOR	
0,1	Null ($00), followed by number of valid data bytes in sector.
2-???	Last bytes of data. Any remaining bytes are garbage and may be ignored.

RELATIVE FILE FORMAT

BYTE	DEFINITION
DATA BLOCK	
0,1	Track and sector of next data block.
2-255	254 bytes of data. Empty records contain $FF (all binary ones) in the first byte followed by $00 (binary all zeros) to the end of the record. Partially filled records are padded with nulls ($00).
SIDE SECTOR BLOCK	
0-1	Track and sector of next side sector block in this group
2	Side sector number (0-5)
3	Record length
4-5	Track and sector of first side sector (number 0)
6-7	Track and sector of second side sector (number 1)
8-9	Track and sector of third side sector (number 2)
10-11	Track and sector of fourth side sector (number 3)
12-13	Track and sector of fifth side sector (number 4)
14-15	Track and sector of sixth side sector (number 5)
16-255	Track and sector pointers to 120 data blocks.
SUPER SIDE SECTOR BLOCK	
0-1	Track and sector of first side sector in group 0
2	$FE
3-4	Track and sector of first side sector in group 0
5-6	Track and sector of first side sector in group 1
253-254	Track and sector of first side sector in group 125

The super side sector has pointers to 126 groups of side sectors. Each of these groups contains 6 side sectors. Each side sector points to 120 data blocks, containing 254 bytes each. 126 × 6 × 120 × 254 = 23,042,880 bytes (maximum relative file size)

APPENDIX D
DISK COMMAND QUICK REFERENCE CHART

General Format: OPEN 15,8,15:PRINT#15,command:CLOSE 15 (Basic 2)

HOUSEKEEPING COMMANDS

BASIC 2.0	NEW	"N0:diskette name,id"
	COPY	"C0:new file=0:old file"
	RENAME	"R0:new name=old name"
	SCRATCH	"S0:file name"
	VALIDATE	"V0"
	INITIALIZE	"IO"
BASIC 7.0/ 3.5	NEW	HEADER"diskette name",Iid,D0
	COPY	COPY "old file" TO "new file"
	RENAME	RENAME "old name" TO "new file"
	SCRATCH	SCRATCH "file name"
	VALIDATE	COLLECT
	INITIALIZE	DCLEAR
BOTH	SELECT PARTITION	"/0:partition name"
	CREATE PARTITION	"/0:partition," CHR$ (starting track) CHR$ (starting sector) CHR$ (<# of blocks) CHR$ (>## of blocks) ",C"

FILE COMMANDS

BASIC 2.0	LOAD	LOAD"file name",8
	SAVE	SAVE"file name",8
	VERIFY	VERIFY"file name",8
BASIC 7.0/ 3.5	LOAD	DLOAD"file name"
	SAVE	DSAVE"file name"
	VERIFY	DVERIFY"file name" (BASIC 7.0 only)
BASIC 7.0 only	BLOAD	BLOAD"filename",Bbank#,Pstart address
	BSAVE	BSAVE"filename",Bbank#,Pst.add To Pen.add
	BOOT	BOOT"filename"
	OPEN	DOPEN#file#,"filename" [.Lrecord length] [,W]
	CLOSE	DCLOSE#file#
	RECORD#	RECORD#file#,record number [,offset]
BOTH	OPEN	OPENfile#,8,channel#,"0:file name,file type, direction"
	CLOSE	CLOSEfile#
	RECORD#	"P"+CHR$(channel#)+CHR$(<record#) +CHR$(>record#)+CHR$(offset)
	PRINT#	PRINT#file#,data list
	GET#	GET#file#,variable list
	INPUT#	INPUT#file#,variable list

DIRECT ACCESS COMMANDS

BLOCK-ALLOCATE	"B-A";0;track#;sector#
BLOCK-EXECUTE	"B-E";channel#;0;track#;sector#
BLOCK-FREE	"B-F";0;track#;sector#
BUFFER-POINTER	"B-P";channel#;byte
BLOCK-READ	"U1";channel#;0;track;sector#
BLOCK-WRITE	"U2";channel#;0;track#;sector#
MEMORY-EXECUTE	"M-E"CHR$(<address)CHR$(>address)
MEMORY-READ	"M-R"CHR$(<address)CHR$(>address)CHR$(# of bytes)
MEMORY-WRITE	"M-W"CHR$(<address)CHR$(>address)CHR$(# of bytes) CHR$(data byte) . . .
USER	"Ucharacter"
UTILITY LOADER	"&0:file name"

FOR MORE DETAILED DESCRIPTIONS OF THESE COMMANDS, CONSULT YOUR COMPUTER'S USER'S GUIDE.

APPENDIX E
SPECIFICATIONS OF THE 1581 DISK DRIVE

STORAGE

Total unformatted capacity	1 Megabyte
Total formatted capacity	808,960 bytes
Maximum Sequential file size	802,640 bytes
Maximum Relative file size	≃ 800K
Records per file	65535
Files per diskette	296
Cylinders per diskette	80
Logical sectors per cylinder	40 (0-39)
Physical sectors per cylinder	20 (1-20)
Free blocks per disk	3160
Logical bytes per sector	256
Physical bytes per sector	512

INTEGRATED CIRCUIT CHIPS USED

6502A	microprocessor
8520A	I/O
23256	32K bytes ROM
4364	8K bytes RAM
WD177X	Floppy Disk Controller

PHYSICAL DIMENSIONS

Height	63 mm
Width	140 mm
Depth	230 mm
Weight	1.4 kg

ELECTRICAL REQUIREMENTS

Voltage	North America	100–120 VAC
	Europe/Australia	220–240 VAC
Frequency	North America	60 Hz
	Europe/Australia	50 Hz
Power used		10 Watts

MEDIA

Any good quality 3.5 inch double-sided diskette may be used.

APPENDIX F
SERIAL INTERFACE INFORMATION

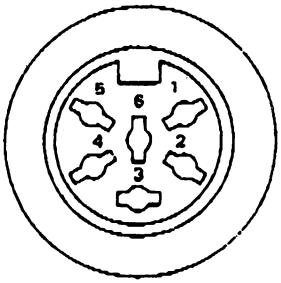
The Serial Interface consists of two 6-pin DIN Female Connectors on each drive. The second connector is for daisy chaining to other drives and/or peripherals. The voltage interface is a serial interface at TTL levels.

There are three types of operation over a serial bus—Control, Talk, and Listen. The host is the controller and initiates all protocol on the serial bus. The host requests the peripheral to listen or talk (if the peripheral is capable of talking as disk drive). All devices connected to the serial bus receive data transmitted over the bus. To allow the host to route its data to an intended destination, each device has a bus address (known as device number). Disk drive's device addresses are normally 8-11.

Data and control signals as follows:

Pin No.	Signal	Direction	Description
Pin 1	SRQ (Service Request)	in/out	Used by fast serial bus as a bi-direction fast clock line. Unused by the slow serial bus.
Pin 2	GND (Ground)		Logic ground
Pin 3	ATN (Attention)	in	The host brings this signal low which then generates an interrupt on the controller board. The attention sequence is followed by a device address. If the device does not respond within a preset time the host will assume the device addressed is not on the bus.
Pin 4	CLK (Clock)	in/out	This signal is used for timing the data sent on slow serial bus (software clocked).
Pin 5	DATA	in/out	Data on the serial bus is transmitted one bit at a time (software toggled in slow mode, hardware toggled in fast mode).
Pin 6	RESET		This line will reset the peripheral upon host reset.

The 6-pin DIN connector looks like (from outside):



In detail, the 1581 serial bus supports the newer FAST serial communication as well as standard (SLOW) serial communication.

The important difference between the FAST serial bus and the SLOW serial bus is the incorporation of the hardware controlled lines for the CLOCK and DATA lines. Fast serial communication is transparent to any peripheral connected to the serial bus that does not contain the necessary hardware or software to talk at fast speed.

To remain compatible with the SLOW serial bus all bytes sent under attention are sent slow.

COMMODORE™

Commodore Business Machines, Inc.
1200 Wilson Drive • West Chester, PA 19380

Commodore Business Machines, Ltd.
3470 Pharmacy Avenue • Agincourt, Ontario, M1W 3G3

Printed in Taiwan

P/N:319928-01

41A048-001

EL MATERIAL EXHIBIDO
ES SOLO PARA USO
EDUCATIVO, NO COMERCIAL

ENCICLOPEDIA DE
REFERENCIA | EDUCATIVA

RETROTECNIA

PARA ENTENDER LA EVOLUCION DE LOS ORDENADORES A TRAVES DEL TIEMPO